AD631871

RADC-TR-66-7, Volume II
Final Report

ADVANCED COMPUTER ORGANIZATION STUDY

Volume II - Appendixes

Donald L. Rohrbacher

TECHNICAL REPORT NO. RADC-TR-66-7
April 1966

Distribution of this document is unlimited

Information Processing Branch
Rome Air Development Center
Research and Technology Division
Air Force Systems Command
Griffiss Air Force Base, New York

# ADVANCED COMPUTER ORGANIZATION STUDY

## Volume II - Appendixes

Donald L. Rohrbacher

# FOREWORD

This technical documentary report records the efforts and achievements on the advanced computer organization study conducted by Goodyear Aerospace Corporation, Akron, Ohio. The secondary report number assigned to this document by the company is GER-12314. This report is published in two volumes: Volume One, Advanced Computer Organization, Basic Report, and Volume Two, Advanced Computer Organization Study, Appendixes.

The study was conducted for the Rome Air Development Center (RADC), Air Force Systems Command, under Contract AF30(602)-3550, Project No. 4594, Task No. 459406. The RADC project monitor was Mr. Fred Dion. The report covers the 14-month period ending 30 November 1965.

Appreciation is extended to Dr. John Holland, University of Michigan, whose consulting services were extremely valuable in both the development and conception of many of the ideas presented. The major contributors to this study were D. L. Rohrbacher (project engineer), Dr. K. E. Batcher, P. A. Gilmore, and G. W. Lahue. Substantial contributions also were made by G. P. Elliott, Dr. C. C. Foster, and D. C. Gilliland.

# TABLE OF CONTENTS

**Page**

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# LIST OF ILLUSTRATIONS

# LIST OF ILLUSTRATIONS

## LIST OF ILLUSTRATIONS

## LIST OF ILLUSTRATIONS

# LIST OF TABLES

# LIST OF TABLES

## LIST OF TABLES

# APPENDIX I - PARALLEL EXECUTION OF
# THE DYNAMIC PROGRAMMING TECHNIQUE

## 1. DYNAMIC PROGRAMMING

Dynamic programming is a mathematical technique devised by Bellman[a] for maximizing a function of n variables:

$$R_n(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} g_i(x_i), \qquad (I-1)$$

where

$$g_i(0) = 0 \text{ and}$$

$$g_i(x_i) \geqq 0$$

over the region

$$S_n(x) = \left\{ (x_1, x_2, \ldots, x_n) \,\Big|\, \sum_{i=1}^{n} x_i = x, \; x_i \geqq 0 \right\}. \qquad (I-2)$$

The dynamic programming technique is directly applicable to allocation problems.

Consider the x of Equation I-2 to be a resource that is to be allocated to some n activities. Let $x_i$ denote the allocation to activity i, and $g_i(x_i)$ the resultant return from activity i. Then the total return from all n activities may be expressed by Equation I-1. The problem is to determine an optimal policy of allocation; that is, to maximize Equation I-1 and

---

[a] Bellman, R. E., and Dreyfus, S. E.: Applied Dynamic Programming. Princeton, N. J., Princeton University Press, 1962.

determine the allocations by which the maximization is effected. The dynamic programming solution to the maximization problem rests on the "discretization" of the range $[0, x]$ and the application of Bellman's principle of optimality, which may be stated: "an optimal policy has the property that whatever the initial state and initial decision are, the remaining decision must constitute an optimal policy with regard to the state resulting from the first decision. "[a]

In the execution of the dynamic programming technique, the following sequence is constructed:

$$f_1(x), \ f_2(x), \ . \ . \ . \ , \ f_n(x) \tag{I-3}$$

where

$$f_k(x) = \max_{S_k(x)} \left[ R_k(x_1, \ x_2, \ . \ . \ . \ , \ x_k) \right] \tag{I-4}$$

with

$R_k(x_1, x_2, \ . \ . \ . \ , x_k)$ and $S_k(x)$ as defined in Equations I-1 and I-2.

Making the reasonable definition,

$$f_0(x) = 0 \tag{I-5}$$

and noting that $f_1(x) = g_1(x)$, the following recursive relation can be deduced:[a]

$$f_k(x) = \max_{0 \leq x_k \leq x} \left[ g_k(x_k) + f_{k-1}(x - x_k) \right] \tag{I-6}$$

thus establishing an inductive method for determining the sequence (I-3). Equation I-6 is just the mathematical expression for the principle of optimality; it allows the reduction of the problem of maximizing one function of n variables to that of maximizing n functions of one variable. In

[a] Ibid.

the execution of the dynamic programming technique, the following se-
quence also is constructed:

$$x_1(x), \; x_2(x), \; \ldots, \; x_n(x),  \qquad \text{(I-7)}$$

where $x_k(x)$ is the allocation to $g_k(x_k)$ that maximized $f_k(x)$.

The heart of the dynamic programming technique, then, is the construc-
tion of the sequences (I-3) and (I-7). As mentioned above, computa-
tional considerations require the discretizing of the range $[0, x]$, say into
the partition

$$0 = t_0 < \; t_1 < \; t_2 < \; \ldots \; < t_n = x \qquad \text{(I-8)}$$

where $t_i = \Delta i$ for some fixed $\Delta$. A partition such as (I-8) often is de-
noted compactly by "$a(\Delta)b$," which is read "from a to b in steps of $\Delta$."
The calculation of the sequences (I-3) and (I-7) over the partition
(I-8) requires that $\{g_i(x)\}$ be calculated over the partition. For illustra-
tion of the dynamic programming technique, consider the maximizing of
the following:

$$R_6(x_1, \; x_2, \; x_3, \; x_4, \; x_5, \; x_6) = x_1 + x_2^2 + x_3^{\frac{1}{2}} + 2 \sin x_4 +$$

$$g_5(x_5) + g_6(x_6). \qquad \text{(I-9)}$$

where

$$g_5(x_5) = \begin{cases} 2x_5 & \text{if } 0 \leq x_5 \leq 1 \\[2mm] 4 - 2x_5 & \text{if } 1 \leq x_5 \leq 2, \end{cases}$$

and

$$g_6(x_6) = 2\lceil x_6 \rceil.$$

subject to the constraint

$$\sum_{i=1}^{6} x_i = x, \ x_i \geqq 0 , \qquad (I-10)$$

with $[x]$ denoting the greatest integer in x.

This is to be done for each $x = 0(0.1)2$; that is, for x from 0 to 2 in steps of C.1. A flow chart for a sequential dynamic programming solution of this problem is given in Figure I-1. For any solution model, it will be necessary to evaluate the set $\{g_i(x_i)\}$ at each point of some partition. For the present problem, the partition $0(0.1)2$ is chosen, and the functional values are recorded in Table I-1. From this table, the sequences $f_1(x)$, $f_2(x)$, . . . , $f_6(x)$ and $x_1(x)$, $x_2(x)$, . . . , $x_6(x)$ can be determined and recorded as in Table I-2. Recalling that $x_i(x)$ is not necessarily unique, note that Table I-1 contains the information necessary to determine all optimal policies as indicated in Table I-2.

Now consider a method for reading out an optimal policy for a given resource from Table I-2. The method is simply this: Given a resource x, $(0 = x = 2)$, select $x_6(x)$, with $x_6$ the allocation for $g_6(x)$. Now select $x_5(x - x_6)$, which is just $x_5$, tne allocation for $g_5(x)$. Next select $x_4(x - x_6 - x_5)$, which is just $x_4$ for $g_4(x)$, and so forth until the allocations $x_1$, $x_2$, . . . , $x_6$ are determined for the activities $g_1(x_1)$, $g_2(x_2)$, . . . , $g_6(x_6)$. A flow chart for the readout method is given in Figure I-2; this chart ignores multiple solution, but all solutions are indicated in Table I-2.

As an example of the readout process, suppose that $x = 2.0$. Then from Table I-2 it can be seen that the following allocations for $(x_1, x_2, x_3, x_4, x_5, x_6)$ yield $f_6(2) = 4.12$ return: (C, 0, 0.1, 0, 0.9, 1.0), (0, 0, 0.1, 0.1, 0.8, 1.0). (0, 0, 0.1, 0.2, 0.7, 1.0), (0, 0, 0.1, 0.3, 0.6, 1.0). This multiplicity of optimal policies is a result of the nonuniqueness of $\{x_i(x)\}$.
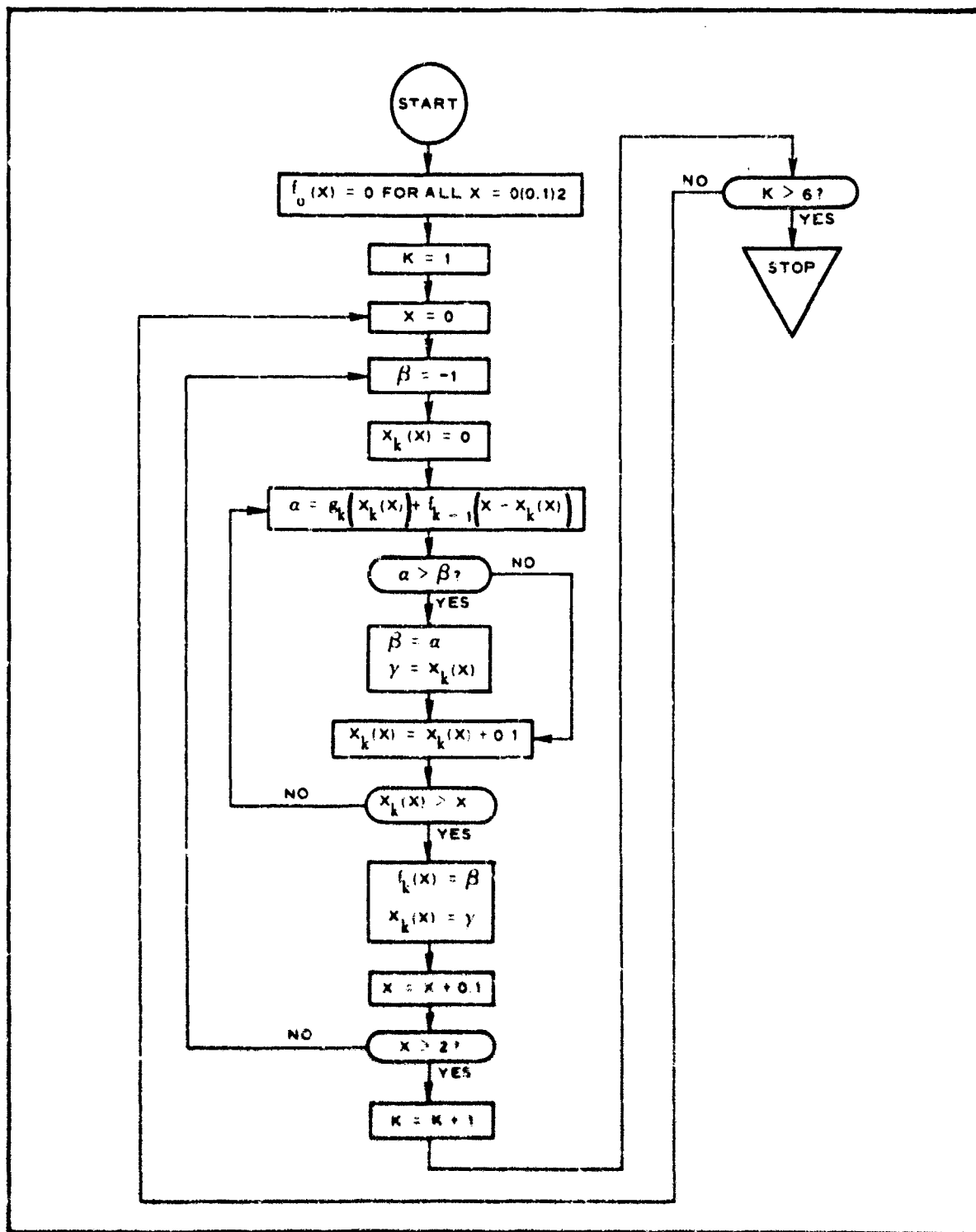
Figure I-1 - Sequential Dynamic Programming Flow Chart

The solution model for the dynamic programming problem considered here specified the calculation of each of the functions $f_1(x)$, $f_2(x)$, . . . ., $f_6(x)$ over the range 0(0.1)2. Hence, if only the first $k$ activities, $k \leqq 6$, are to be considered, the optimal policy for a given x can be readout out easily from Table I-2. For example, let $x = 1.5$ and $k = 4$. Now from Table I-2, $f_4(1.5) = 2.43$, which is achieved by the following allocations for $(x_1, x_2, x_3, x_4)$: (0.3, 0, 0.2, 1.0), (0.2, 0, 0.3, 1.0), (0.2, 0, 0.2, 1.1), and (0.1, 0, 0.3, 1.1).

The solution model described above and outlined in Figure I-1 for the solution of an optimization problem by dynamic programming is sequential in nature. It involved computing first $g_1(t)$ for $t = 0(0.1)2$, then

## TABLE I-1 - ACTIVITY RETURNS FOR EQUATION I-9

| $x$ | $g_1(x) = x$ | $g_2(x) = x^2$ | $g_3(x) = x^{\frac{1}{2}}$ | $g_4(x) = 2 \sin x$ | $g_5(x) = \begin{cases} 2x, & 0 \leqq x \leqq 1 \\ 4 - 2x, & 1 \leqq x \leqq 2 \end{cases}$ | $g_6(x) = 2[x]$ |
|---|---|---|---|---|---|---|
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 0.1 | 0.1 | 0.01 | 0.32 | 0.20 | 0.2 | 0.0 |
| 0.2 | 0.2 | 0.04 | 0.45 | 0.40 | 0.4 | 0.0 |
| 0.3 | 0.3 | 0.09 | 0.55 | 0.60 | 0.6 | 0.0 |
| 0.4 | 0.4 | 0.16 | 0.63 | 0.78 | 0.8 | 0.0 |
| 0.5 | 0.5 | 0.25 | 0.71 | 0.96 | 1.0 | 0.0 |
| 0.6 | 0.6 | 0.36 | 0.77 | 1.14 | 1.2 | 0.0 |
| 0.7 | 0.7 | 0.49 | 0.84 | 1.28 | 1.4 | 0.0 |
| 0.8 | 0.8 | 0.64 | 0.89 | 1.44 | 1.6 | 0.0 |
| 0.9 | 0.9 | 0.81 | 0.95 | 1.56 | 1.8 | 0.0 |
| 1.0 | 1.0 | 1.00 | 1.00 | 1.68 | 2.0 | 2.0 |
| 1.1 | 1.1 | 1.21 | 1.05 | 1.78 | 1.8 | 2.0 |
| 1.2 | 1.2 | 1.44 | 1.10 | 1.86 | 1.6 | 2.0 |
| 1.3 | 1.3 | 1.69 | 1.14 | 1.92 | 1.4 | 2.0 |
| 1.4 | 1.4 | 1.96 | 1.18 | 1.98 | 1.2 | 2.0 |
| 1.5 | 1.5 | 2.25 | 1.22 | 2.00 | 1.0 | 2.0 |
| 1.6 | 1.6 | 2.56 | 1.26 | 2.00 | 0.8 | 2.0 |
| 1.7 | 1.7 | 2.89 | 1.30 | 1.98 | 0.6 | 2.0 |
| 1.8 | 1.8 | 3.24 | 1.34 | 1.94 | 0.4 | 2.0 |
| 1.9 | 1.9 | 3.61 | 1.38 | 1.90 | 0.2 | 2.0 |
| 2.0 | 2.0 | 4.00 | 1.41 | 1.82 | 0.0 | 4.0 |

$g_2(t)$ for $t = 0(0.1)2$, and so forth until $g_6(t)$ was computed. These results make up Table I-1. From this table, the sequences $f_1(x)$, $f_2(x)$, ..., $f_6(x)$ and $x_1(x)$, $x_2(x)$, ..., $x_6(x)$ were computed sequentially for $x = 0(0.1)2$

Finally, a readout process was specified for determining optimal allocations for a given resource. The whole solution model was sequential, but the method of dynamic programming itself is not essentially sequential in nature.

Now it will be indicated how parallel aspects of the dynamic programming method may be exploited in a solution model similar to the one described above.

### TABLE I-2 - SEQUENTIAL MAXIMIZATION OF EQUATION I-9

| x | $x_1(x)$ | $f_1(x)$ | $x_2(x)$ | $f_2(x)$ | $x_3(x)$ | $f_3(x)$ | $x_4(x)$ | $f_4(x)$ | $x_5(x)$ | $f_5(x)$ | $x_6(x)$ | $f_6(x)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.0 | 0.0 | 0 0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0 0 | 0.0 | 0 0 |
| 0.1 | 0.1 | 0 1 | 0.0 | 0 1 | 0 1 | 0 32 | 0 0 | 0 32 | 0 0 | 0 32 | 0 0 | 0 32 |
| 0.2 | 0.2 | 0 2 | 0 0 | 0.2 | 0 2 | 0 45 | 0 1 | 0 52 | 0, 0.1 | 0 52 | 0.0 | 0 52 |
| 0.3 | 0.3 | 0 3 | 0 0 | 0 3 | 0.2  0 3 | 0 55 | 0 2 | 0 72 | 0, 0 1, 0 2 | 0 72 | 0.0 | 0 72 |
| 0.4 | 0.4 | 0 4 | 0 0 | 0 4 | 0 2, 0 3 | 0 65 | 0 3 | 0 82 | 0, 0 1, 0 2, 0 3 | 0 92 | 0.0 | 0 92 |
| 0.5 | 0.5 | 0 5 | 0 0 | 0 5 | 0 2, 0 3 | 0 75 | 0 4 | 1 05 | 0 1, 0 2, 0 3, 0 4 | 1 12 | 0 0 | 1 12 |
| 0.6 | 0 6 | 0 6 | 0 0 | 0 6 | 0 2, 0 3 | 0 85 | 0 5 | 1 28 | 0 2, 0 3, 0 4, 0 5 | 1 32 | 0 0 | 1 32 |
| 0 7 | 0 7 | 0 7 | 0 0 | 0 7 | 0 2, 0 3 | 0 95 | 0 6 | 1 46 | 0 3, 0 4, 0 5, 0 6 | 1 52 | 0 0 | 1 52 |
| 0 8 | 0 8 | 0 8 | 0 0 | 0 8 | 0 2, 0 3 | 1 05 | 0 7 | 1 60 | 0 4, 0 5, 0 6, 0 7 | 1 72 | 0 0 | 1 72 |
| 0 9 | 0 9 | 0 9 | 0 0 | 0 9 | 0 2, 0 3 | 1 15 | 0 8 | 1 76 | 0 5, 0 6, 0 7, 0 8 | 1 92 | 0 0 | 1 92 |
| 1 0 | 1 0 | 1 0 | 0 1 | 1 0 | 0 2, 0 3 | 1 25 | 0 8 | 1 89 | 0 6, 0 7, 0 8, 0 9 | 2 12 | 0 0 | 2 12 |
| 1 1 | 1 1 | 1 1 | 1 1 | 1 21 | 0 2, 0 3 | 1 35 | 0 9 | 2 01 | 0 7, 0 8, 0 9, 1 0 | 2 32 | 0, 1 0 | 2 32 |
| 1 2 | 1 2 | 1 2 | 1 2 | 1 44 | 0 1 | 1 53 | 1 0 | 2 13 | 0 8, 0 9, 1 0 | 2 52 | 0, 1 0 | 2 52 |
| 1 3 | 1 3 | 1 3 | 1 3 | 1 69 | 0 1 | 1 76 | 1 0, 1 1 | 2 23 | 0 9, 1 0 | 2 72 | 0, 1 0 | 2 72 |
| 1 4 | 1 4 | 1 4 | 1 4 | 1 96 | 0 1 | 2 01 | 1 0, 1 1 | 2 33 | 1 0 | 2 92 | 0, 1 0 | 2 92 |
| 1 5 | 1 5 | 1 5 | 1 5 | 2 25 | 0 1 | 2 28 | 1 0, 1 1 | 2 43 | 0 9 | 3 04 | 1 0 | 3 12 |
| 1 6 | 1 6 | 1 6 | 1 6 | 2 56 | 0 1 | 2 57 | 2 0 | 2 57 | 1 0 | 3 28 | 1 0 | 3 32 |
| 1 7 | 1 7 | 1 7 | 1 7 | 2 89 | 0 0 | 2 89 | 2 0 | 2 89 | 1 0 | 3 46 | 1 0 | 3 52 |
| 1 8 | 1 8 | 1 8 | 1 8 | 3 24 | 0 0 | 3 24 | 2 0 | 3 24 | 1 0 | 3 60 | 1 0 | 3 72 |
| 1 9 | 1 9 | 1 9 | 1 9 | 3 61 | 0 0 | 3 61 | 2 0 | 3 61 | 1 0 | 3 76 | 1 0 | 3 92 |
| 2 0 | 2 0 | 2 0 | 2 0 | 4 00 | 0 0 | 4 00 | 2 0 | 4 00 | 0 0 | 4 00 | 1 0 | 4 12 |
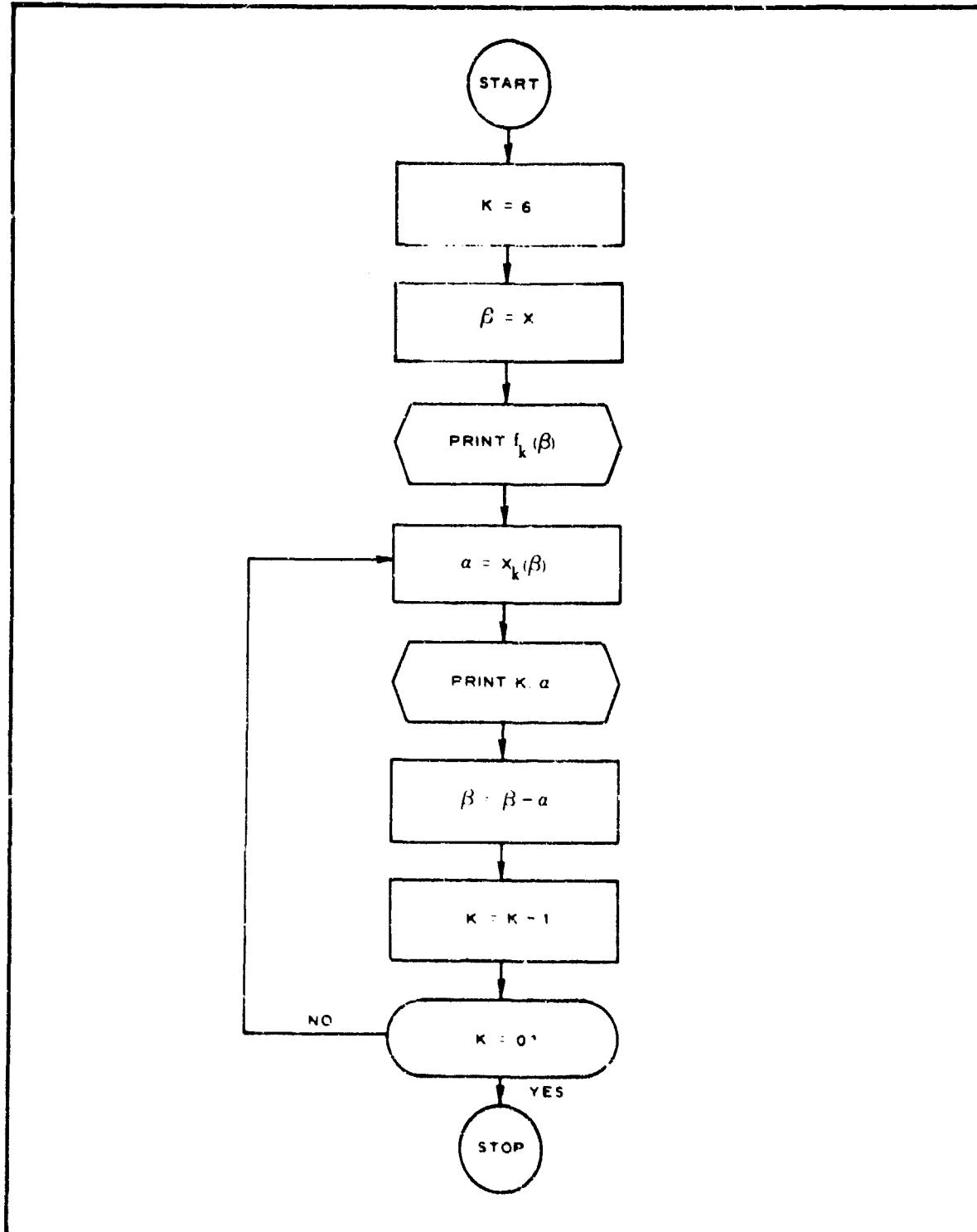
Figure  I-2 - Sequential Optimal Allocation Readout

## 2. PARALLEL SOLUTION MODEL

Consider the problem of maximizing, by dynamic programming techniques, the function

$$R_{2N}(x_1, x_2, \ldots, x_{2N}) = \sum_{i=1}^{2N} g_i(x_i), \qquad (I-11)$$

where $2N$ variables are assumed for convenience. Let

$$x_1 + x_2 + \ldots x_N = y_1 \qquad (I-12)$$

$$x_{N+1} + x_{N+2} + \ldots x_{2N} = y_2 \qquad (I-13)$$

with

$$y_1 + y_2 = x ;$$

$$U_N(y_1) = \left\{ \max_{(x_1, x_2, \ldots x_N) \;\middle|\; \sum_{i=1}^{N} x_i = y_1} \right\} \left[ g_1(x_1) + g_2(x_2) + \ldots + g_N(x_N) \right].$$

$$(I-14)$$

$$V_N(y_2) = \left\{ \max_{(x_{N+1}, x_{N+2}, \ldots x_{2N}) \;\middle|\; \sum_{i=N+1}^{2N} x_i = y_2} \right\}$$

$$\left[ g_{N+1}(x_{N+1}) + g_{N+2}(x_{N+2}) + \ldots g_{2N}(x_{2N}) \right] \qquad (I-15)$$

Now $U_N(y_1)$ and $V_N(y_2)$ may be computed independently and thus in parallel, using Equation I-6. This equation then can be used to maximize the sum $U_N(y_1) + V_N(y_2)$ over

$$\left| (y_1, y_2) \,\middle|\, y_1 + y_2 = x \right|.$$

Since computer time is approximately proportional to the number of variables, the sequential solution time for the maximization of Equation I-11 is proportional to 2N; the corresponding parallel solution time is proportional to N + 2. If $2^N$ variables were involved, the $2^{N-1}$ pairs first could be processed in parallel, then the resulting $2^{N-2}$ pairs, and so on. The sequential solution time for the case of $2^N$ variables would be proportional to $2^N$; the parallel solution time would be proportional to 2N.

In addition to parallel aspects of the maximization of (I-11), parallelism exists at the lowest-level computations (fundamental and subroutine type computations). For example, initially the 2N vectors $\left[g_i(0), g_i(\Delta), g_i(2\Delta), \ldots g_i(x)\right]$ $i = 1, 2, \ldots, 2N$ can be computed in parallel.

As a specific example of the injection of parallelism into a dynamic programming solution model, consider now the example problem introduced above; that is, the maximization of (I-9) under the constraint (I-10). The several activity functions of Equation I-9 are independent of one another and hence can be calculated in parallel on a parallel processor. However, efficient use of a parallel processor prohibits parallel computations that contribute little or nothing to improved solution speeds, and hence tax machine capacity unnecessarily. Since the calculation of the sequences (I-3) and (I-7) is the ultimate goal of the dynamic programming technique, values for the activity functions $\{g_i(t)\}$ need not be calculated prior to the time when the values are needed in the computation of (I-3) and (I-7).

Consider a partition $x_0(\Delta)x$ to be used in the dynamic programming maximization of a function of type (I-1).

In general,

$$f_k(x_0 + j\Delta) = \max_{0 \leq i \leq j}\left[g_k(x + i\Delta) + f_{k-1}\left(x + (j - i)\Delta\right)\right]. \qquad (I-16)$$

Hence, to calculate $f_k(x_0 + j\Delta)$ only the following values must be known:

$$f_{k-1}(x_0), \ f_{k-1}(x_0 + \Delta), \ \ldots, \ f_{k-1}(x_0 + j\Delta)$$

and
$$\left. \begin{array}{c} \\ \\ \end{array} \right\} \quad \text{(I-17)}$$

$$g_k(x_0), \ g_k(x_0 + \Delta), \ \ldots, \ g_k(x_0 + j\Delta)$$

and the calculation of the sequences (I-3) and (I-7) can in fact be carried on in parallel with the calculation of the activity functions $\{g_i(t)\}$.

Now for the parallel execution of the dynamic programming maximization of (I-9), make the following definitions:

$$u_1(x) = \max_{0 = y \leq x} \left[ g_2(y) + g_1(x - y) \right],$$

$$y_1(x) = y \text{ at which the maximum occurs;} \qquad \text{(I-18)}$$

$$u_2(x) = \max_{0 \leq y \leq x} \left[ g_4(y) + g_3(x - y) \right],$$

$$y_2(x) = y \text{ at which the maximum occurs;} \qquad \text{(I-19)}$$

$$u_3(x) = \max_{0 \leq y \leq x} \left[ g_6(y) + g_5(x - y) \right],$$

$$y_3(x) = y \text{ at which the maximum occurs;} \qquad \text{(I-20)}$$

$$u_4(x) = \max_{0 \leq y \leq x} \left[ u_2(y) + u_1(x - y) \right],$$

$$y_4(x) = y \text{ at which the maximum occurs;} \qquad \text{(I-21)}$$

$$u_5(x) = \max_{0 \leq y \leq x} \left[ u_4(y) + u_3(x - y) \right],$$

$$y_5(x) = y \text{ at which the maximum occurs.} \qquad \text{(I-22)}$$

Consider the partition $0(0.1)2$ in terms of $x_0 = 0$ and $\Delta = 0.1$, and then $x_i = x_0 + i\Delta$, $i = 0, 1, 2, \ldots, 20$. A chart can be constructed showing the level-by-level parallel execution of the dynamic programming

maximization of (I-9), as in Table I-3. It will be noted that in Table I-3 a level corresponds to a new stage of computation for the activity functions $\{g_i(t)\}$ and return functions $\{u_i(t)\}$ over the partition $t = x_0(\Delta)x = 0(0.1)2$. Computation of the sequence $y_i(t)$, $i = 1, 2, \ldots,$ 5 has not been indicated, but the required values are an immediate consequence of the calculation of the sequence $u_i(t)$, $i = 1, 2, \ldots, 5$.

As shown in Table I-3, the partition $x_0(\Delta)x$ and activity functions $g_i(x)$, $i = 1, 2, \ldots, 6$ over the partition can be calculated in 21 levels through parallel computation. The same computations, performed in a sequential manner, would require 141 levels. A further indication of the parallel characteristics of the dynamic programming technique and the power of parallel processing is seen in that only three additional levels of computation allow the complete maximization of (I-9) to be effected. Sequential techniques would require 110 additional levels of computation. Hence parallel techniques offer a total advantage of 24 to 251 for the problem at hand. The difference in computational levels required by parallel and sequential models indicated here, striking as it is, only begins to point out the increased computational speed offered by parallel execution of the dynamic programming technique, since no appeal has been made to parallel execution of basic machine instructions effecting the individual computational levels.

The results of the parallel dynamic programming computation for Equation I-9 are given in Table I-4. A generalized readout process is given in Figure I-3. A specific readout for $x = 2.0$ is given in Figure I-4.

In Table I-4, the maximum possible return of $u_5(2) = 4.12$ is achieved for the following allocations to $(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 0, 0.1, 0.3, 0.6, 1.0)$, $(0, 0, 0.1, 0.2, 0.7, 1.0)$, $(0, 0, 0.1, 0.1, 0.8, 1.0)$, and $(0, 0, 0.1, 0, 0.9, 1.0)$. These allocations agree, of course, with those from the sequential computation.

## TABLE I-3 - PARALLEL SOLUTION MODEL

| Level | Partition | $g_1(x)$* | $g_2(x)$ | $g_3(x)$ | $g_4(x)$ | $g_5(x)$ | $g_6(x)$ | $u_1(x)$+ | $u_2(x)$ | $u_3(x)$ | $u_4(x)$ | $u_5(x)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | $x_1 = x_0 + \Delta$ | $g_1(x_0)$ | $g_2(x_0)$ | $g_3(x_0)$ | $g_4(x_0)$ | . . . | . . . | . . . | . . . | . . . | . . . | . . . |
| 1 | $x_2 = x_1 + \Delta$ | $g_1(x_1)$ | $g_2(x_1)$ | $g_3(x_1)$ | $g_4(x_1)$ | $g_5(x_0)$ | $g_6(x_0)$ | $u_1(x_0)$ | $u_2(x_0)$ | . . . | . . . | . . . |
| 2 | $x_3 = x_2 + \Delta$ | $g_1(x_2)$ | $g_2(x_2)$ | $g_3(x_2)$ | $g_4(x_2)$ | $g_5(x_1)$ | $g_6(x_1)$ | $u_1(x_1)$ | $u_2(x_1)$ | $u_3(x_0)$ | $u_4(x_0)$ | . . . |
| 3 | $x_4 = x_3 + \Delta$ | $g_1(x_3)$ | $g_2(x_3)$ | $g_3(x_3)$ | $g_4(x_3)$ | $g_5(x_2)$ | $g_6(x_2)$ | $u_1(x_2)$ | $u_2(x_2)$ | $u_3(x_1)$ | $u_4(x_1)$ | $u_5(x_0)$ |
| 4 | $x_5 = x_4 + \Delta$ | $g_1(x_4)$ | $g_2(x_4)$ | $g_3(x_4)$ | $g_4(x_4)$ | $g_5(x_3)$ | $g_6(x_3)$ | $u_1(x_3)$ | $u_2(x_3)$ | $u_3(x_2)$ | $u_4(x_2)$ | $u_5(x_1)$ |
| 5 | $x_6 = x_5 + \Delta$ | $g_1(x_5)$ | $g_2(x_5)$ | $g_3(x_5)$ | $g_4(x_5)$ | $g_5(x_4)$ | $g_6(x_4)$ | $u_1(x_4)$ | $u_2(x_4)$ | $u_3(x_3)$ | $u_4(x_3)$ | $u_5(x_2)$ |
| 6 | $x_7 = x_6 + \Delta$ | $g_1(x_6)$ | $g_2(x_6)$ | $g_3(x_6)$ | $g_4(x_6)$ | $g_5(x_5)$ | $g_6(x_5)$ | $u_1(x_5)$ | $u_2(x_5)$ | $u_3(x_4)$ | $u_4(x_4)$ | $u_5(x_3)$ |
| 7 | $x_8 = x_7 + \Delta$ | $g_1(x_7)$ | $g_2(x_7)$ | $g_3(x_7)$ | $g_4(x_7)$ | $g_5(x_6)$ | $g_6(x_6)$ | $u_1(x_6)$ | $u_2(x_6)$ | $u_3(x_5)$ | $u_4(x_5)$ | $u_5(x_4)$ |
| 8 | $x_9 = x_8 + \Delta$ | $g_1(x_8)$ | $g_2(x_8)$ | $g_3(x_8)$ | $g_4(x_8)$ | $g_5(x_7)$ | $g_6(x_7)$ | $u_1(x_7)$ | $u_2(x_7)$ | $u_3(x_6)$ | $u_4(x_6)$ | $u_5(x_5)$ |
| 9 | $x_{10} = x_9 + \Delta$ | $g_1(x_9)$ | $g_2(x_9)$ | $g_3(x_9)$ | $g_4(x_9)$ | $g_5(x_8)$ | $g_6(x_8)$ | $u_1(x_8)$ | $u_2(x_8)$ | $u_3(x_7)$ | $u_4(x_7)$ | $u_5(x_6)$ |
| 10 | $x_{11} = x_{10} + \Delta$ | $g_1(x_{10})$ | $g_2(x_{10})$ | $g_3(x_{10})$ | $g_4(x_{10})$ | $g_5(x_9)$ | $g_6(x_9)$ | $u_1(x_9)$ | $u_2(x_9)$ | $u_3(x_8)$ | $u_4(x_8)$ | $u_5(x_7)$ |
| 11 | $x_{12} = x_{11} + \Delta$ | $g_1(x_{11})$ | $g_2(x_{11})$ | $g_3(x_{11})$ | $g_4(x_{11})$ | $g_5(x_{10})$ | $g_6(x_{10})$ | $u_1(x_{10})$ | $u_2(x_{10})$ | $u_3(x_9)$ | $u_4(x_9)$ | $u_5(x_8)$ |
| 12 | $x_{13} = x_{12} + \Delta$ | $g_1(x_{12})$ | $g_2(x_{12})$ | $g_3(x_{12})$ | $g_4(x_{12})$ | $g_5(x_{11})$ | $g_6(x_{11})$ | $u_1(x_{11})$ | $u_2(x_{11})$ | $u_3(x_{10})$ | $u_4(x_{10})$ | $u_5(x_9)$ |
| 13 | $x_{14} = x_{13} + \Delta$ | $g_1(x_{13})$ | $g_2(x_{13})$ | $g_3(x_{13})$ | $g_4(x_{13})$ | $g_5(x_{12})$ | $g_6(x_{12})$ | $u_1(x_{12})$ | $u_2(x_{12})$ | $u_3(x_{11})$ | $u_4(x_{11})$ | $u_5(x_{10})$ |
| 14 | $x_{15} = x_{14} + \Delta$ | $g_1(x_{14})$ | $g_2(x_{14})$ | $g_3(x_{14})$ | $g_4(x_{14})$ | $g_5(x_{13})$ | $g_6(x_{13})$ | $u_1(x_{13})$ | $u_2(x_{13})$ | $u_3(x_{12})$ | $u_4(x_{12})$ | $u_5(x_{11})$ |
| 15 | $x_{16} = x_{15} + \Delta$ | $g_1(x_{15})$ | $g_2(x_{15})$ | $g_3(x_{15})$ | $g_4(x_{15})$ | $g_5(x_{14})$ | $g_6(x_{14})$ | $u_1(x_{14})$ | $u_2(x_{14})$ | $u_3(x_{13})$ | $u_4(x_{13})$ | $u_5(x_{12})$ |
| 16 | $x_{17} = x_{16} + \Delta$ | $g_1(x_{16})$ | $g_2(x_{16})$ | $g_3(x_{16})$ | $g_4(x_{16})$ | $g_5(x_{15})$ | $g_6(x_{15})$ | $u_1(x_{15})$ | $u_2(x_{15})$ | $u_3(x_{14})$ | $u_4(x_{14})$ | $u_5(x_{13})$ |
| 17 | $x_{18} = x_{17} + \Delta$ | $g_1(x_{17})$ | $g_2(x_{17})$ | $g_3(x_{17})$ | $g_4(x_{17})$ | $g_5(x_{16})$ | $g_6(x_{16})$ | $u_1(x_{16})$ | $u_2(x_{16})$ | $u_3(x_{15})$ | $u_4(x_{15})$ | $u_5(x_{14})$ |
| 18 | $x_{19} = x_{18} + \Delta$ | $g_1(x_{18})$ | $g_2(x_{18})$ | $g_3(x_{18})$ | $g_4(x_{18})$ | $g_5(x_{17})$ | $g_6(x_{17})$ | $u_1(x_{17})$ | $u_2(x_{17})$ | $u_3(x_{16})$ | $u_4(x_{16})$ | $u_5(x_{15})$ |
| 19 | $x_{20} = x_{19} + \Delta$ | $g_1(x_{19})$ | $g_2(x_{19})$ | $g_3(x_{19})$ | $g_4(x_{19})$ | $g_5(x_{18})$ | $g_6(x_{18})$ | $u_1(x_{18})$ | $u_2(x_{18})$ | $u_3(x_{17})$ | $u_4(x_{17})$ | $u_5(x_{16})$ |
| 20 | . . . | $g_1(x_{20})$ | $g_2(x_{20})$ | $g_3(x_{20})$ | $g_4(x_{20})$ | $g_5(x_{19})$ | $g_6(x_{19})$ | $u_1(x_{19})$ | $u_2(x_{19})$ | $u_3(x_{18})$ | $u_4(x_{18})$ | $u_5(x_{17})$ |
| 21 | . . . | . . . | . . . | . . . | . . . | $g_5(x_{20})$ | $g_6(x_{20})$ | $u_1(x_{20})$ | $u_2(x_{20})$ | $u_3(x_{19})$ | $u_4(x_{19})$ | $u_5(x_{18})$ |
| 22 | . . . | . . . | . . . | . . . | . . . | . . . | . . . | . . . | . . . | $u_3(x_{20})$ | $u_4(x_{20})$ | $u_5(x_{19})$ |
| 23 | . . . | . . . | . . . | . . . | . . . | . . . | . . . | . . . | . . . | . . . | . . . | $u_5(x_{20})$ |

*$g_i(x)$ (with $i = 1, 2, \ldots, 6$) defined as on Page 3.

+$u_i(x)$ (with $i = 1, 2, \ldots, 5$) defined as on Page 11.

For a resource of $x = 1.6$, the maximum return of $u_5(1.6) = 2.57$ occurs for the allocations (0, 1.5, 0, 1, 0) if only the first four activities are considered.

## 3. CONCLUSIONS

Dynamic programming has been introduced and illustrated by a specific example. The dynamic programming technique was examined for sequential and parallel characteristics. Parallel characteristics were noted and found to provide a basis for significant increases in processing

TABLE I-4 - PARALLEL MAXIMIZATION OF EQUATION I-9

| x | $y_1(x)$ | $u_1(x)$ | $y_2(x)$ | $u_2(x)$ | $y_3(x)$ | $u_3(x)$ | $y_4(x)$ | $u_4(x)$ | $y_5(x)$ | $u_5(x)$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0.0 | 0.0 |      | 0.0 | 0.0  | 0 0     | 0.0 | 0.0          | 0.0  | 0.0                    | 0 0  |
| 0.1 | 0.0 | 0.1  | 0.0 | 0.32 | 0 0     | 0.2 | 0.1          | 0.32 | 0.1                    | 0 32 |
| 0.2 | 0.0 | 0.2  | 0.1 | 0.52 | 0.0     | 0.4 | 0.2          | 0.52 | 0.2, 0.               | 0.52 |
| 0.3 | 0 0 | 0.3  | 0.2 | 0.72 | 0.0     | 0.6 | 0 3          | 0.72 | 0.2, 0.3, 0.4          | 0.72 |
| 0.4 | 0.0 | 0.4  | 0.3 | 0 92 | 0.0     | 0.8 | 0.4          | 0.92 | 0 1, 0.2, 0 3, 0 4     | 0.92 |
| 0.5 | 0.0 | 0.5  | 0.4 | 1.10 | 0 0     | 1.0 | 0 5          | 1.10 | 0.1, 0.2, 0.3, 0 4     | 1 12 |
| 0.6 | 0.0 | 0.6  | 0.5 | 1.28 | 0.0     | 1.2 | 0 6          | 1.28 | 0.1, 0.2, 0.3, 0.4     | 1 32 |
| 0.7 | 0.0 | 0.7  | 0.6 | 1.46 | 0.0     | 1.4 | 0.7          | 1.46 | 0.1, 0.2, 0 3, 0.4     | 1 52 |
| 0.8 | 0.0 | 0.8  | 0.7 | 1.60 | 0.0     | 1.6 | 0.8          | 1 60 | 0.1, 0.2, 0.3, 0.4     | 1 72 |
| 0.9 | 0.0 | 0.9  | 0.8 | 1.76 | 0.0     | 1.8 | 0.9          | 1 76 | 0 1, 0.2, 0.3, 0.4     | 1.92 |
| 1.0 | 0, 1.0 | 1.0 | 0.8 | 1.89 | 0, 1.0 | 2.0 | 1.0          | 1 89 | 0.1, 0 2, 0.3, 0.4     | 2 12 |
| 1.1 | 1.1 | 1.21 | 0.9 | 2.01 | 1.0     | 2.2 | 1.1          | 2.01 | 0.1, 0.2, 0.3, 0.4     | 2 32 |
| 1.2 | 1.2 | 1.44 | 1.0 | 2.13 | 1.0     | 2.4 | 1.2          | 2.13 | 0 1, 0.2, 0.3, 0.4     | 2.52 |
| 1.3 | 1.3 | 1.69 | 1.0, 1.1 | 2.23 | 1.0 | 2 6 | 1.2, 1.3     | 2.23 | 0 1, 0.2, 0.3, 0 4     | 2 72 |
| 1.4 | 1.4 | 1.96 | 1.1 | 2.33 | 1.0     | 2.8 | 1.2, 1.3, 1.4 | 2 33 | 0 1, 0.2, 0.3, 0 4     | 2.92 |
| 1.5 | 1.5 | 2.25 | 1.1, 1.2 | 2.41 | 1 0 | 3.0 | 1.2, 1.3, 1.4 | 2.43 | 0.1, 0.2, 0 3, 0.4     | 3 12 |
| 1.6 | 1.6 | 2.56 | 1.1, 1.2 | 2.49 | 1.0 | 3.2 | 0 1          | 2 57 | 0. , 0.2, 0.3, 0 4     | 3 32 |
| 1 7 | 1.7 | 2.89 | 1.2 | 2.57 | 1.0     | 3.4 | 0.0          | 2 89 | 0 1, 0.2, 0.3, 0.4     | 3.52 |
| 1.8 | 1.8 | 3.24 | 1.2, 1.3 | 2.63 | 1 0 | 3 6 | 0.0          | 3 24 | 0.1, 0.2, 0.3, 0.4     | 3.72 |
| 1.9 | 1.9 | 3.61 | 1.2 | 2 70 | 1 0     | 3.8 | 0 0          | 3.61 | 0.1, 0 2, 0.3, 0 4     | 3.92 |
| 2.0 | 2.0 | 4.00 | 1.3 | 2.76 | 1 0, 2 0 | 4 0 | 0.0         | 4.00 | 0.1, 0.2, 0 3, 0.4     | 4.12 |

time. The results indicate that construction of efficient solution models for parallel processors depends heavily on analysis of the problem and machine at hand, so that machine capacity is not unnecessarily taxed in parallel computations that improve solution speeds very little if at all.
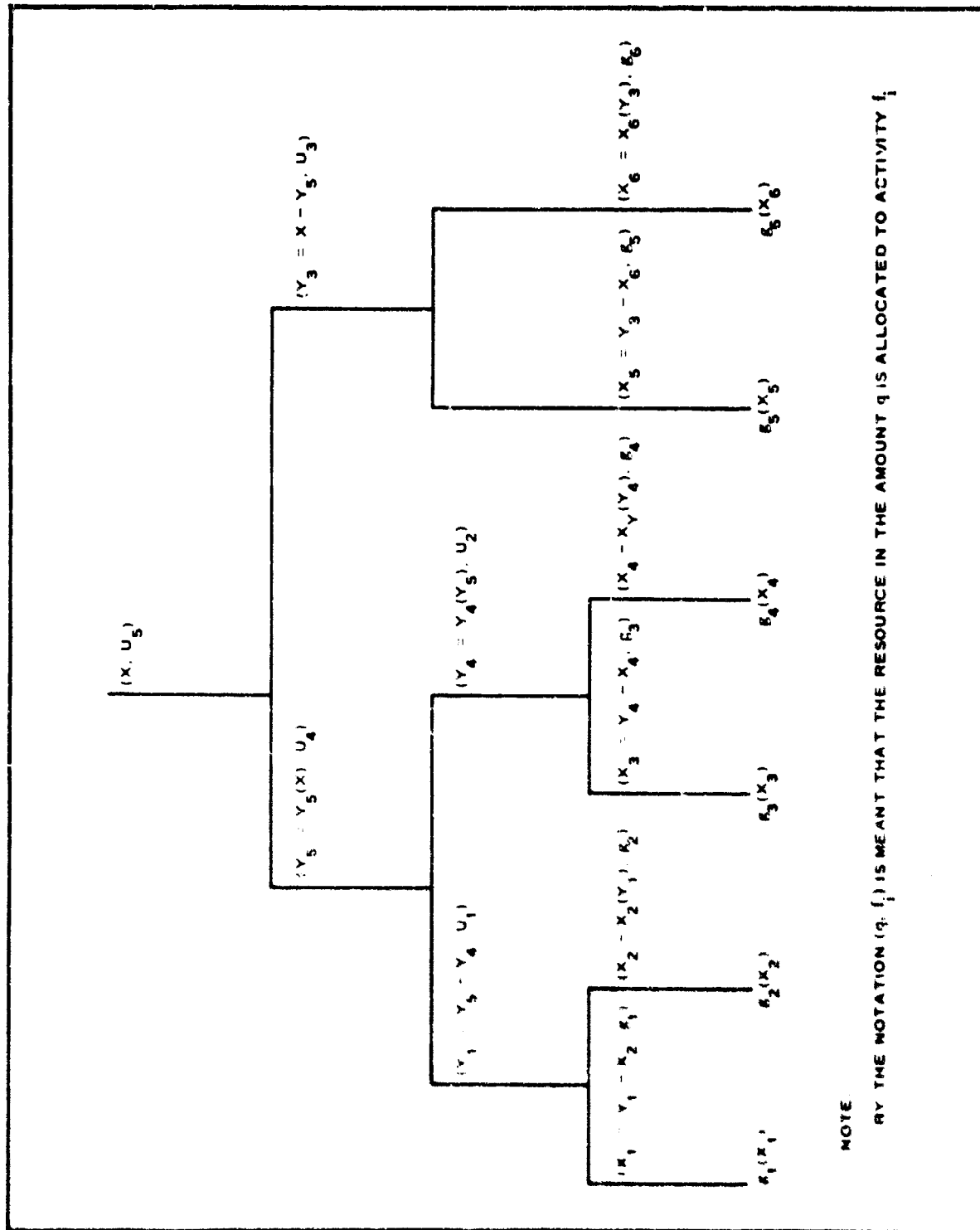
Figure I-3 - Readout Process for Parallel Solution Model
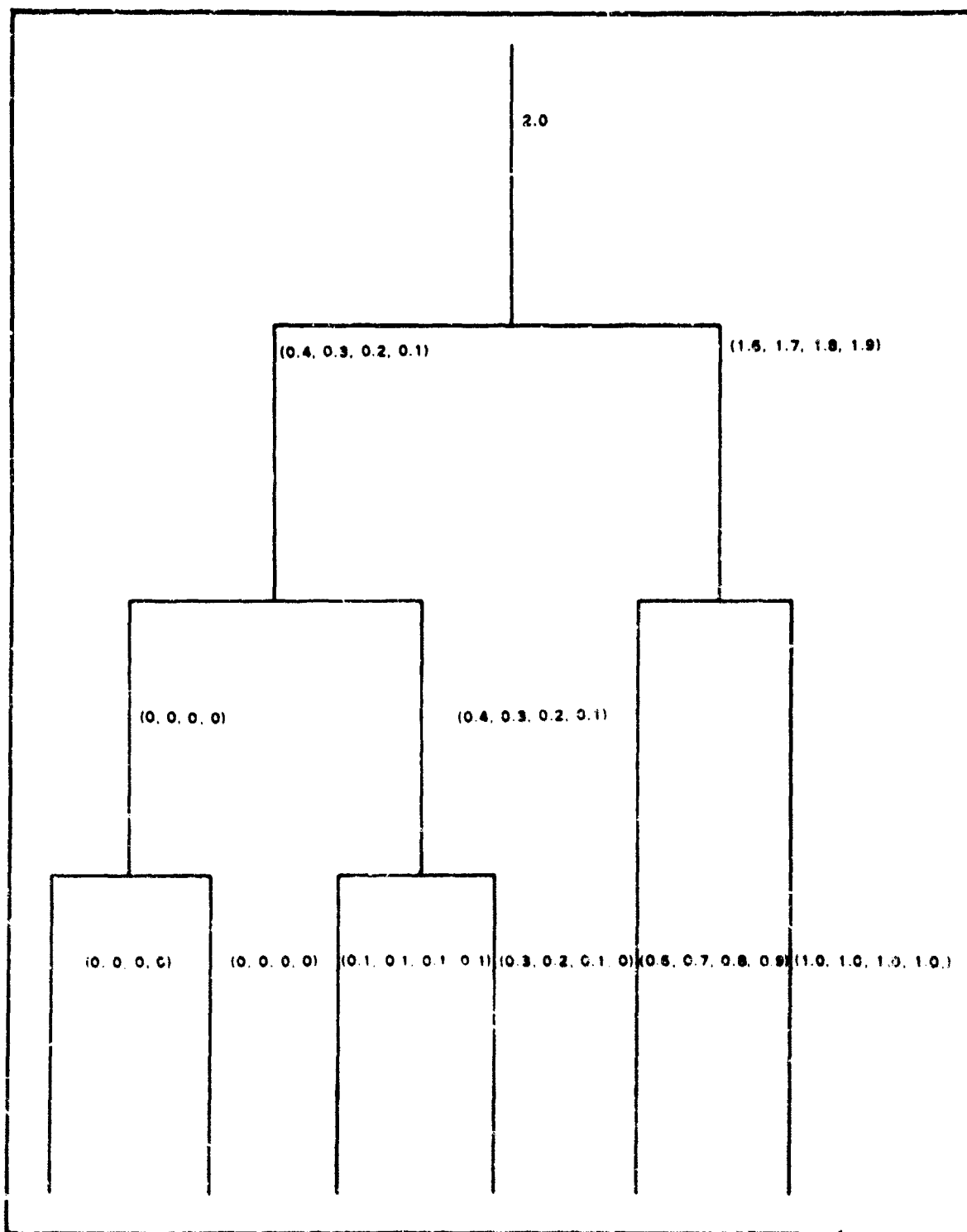
Figure I-4 - Readout Process for x = 2.0

# APPENDIX II - PROGRAMMING OF THE DYNAMIC PROGRAMMING

## TECHNIQUE FOR THE IBM 7090 (SEQUENTIAL)

1. INTRODUCTION

A dynamic programming problem was programmed for a standard general-purpose computer and for Machine I to compare the operation of the two computers. The IBM 7090 was chosen as the standard.

In programming the problem, no input-output operations are performed. The program is assumed to be available in storage. The results are stored in tables according to the table layout diagram (see page 42). To obtain a recommended resource assignment for a given number of activities, the input data (N), the number of activities to be considered, and the quantity of resource to be assigned $(X_o)$, are assumed to be in storage prior to starting the lookup routine. After the lookup routine is executed, the recommended resource assignment to each activity is found in the $\theta$ output table.

Minimum and maximum program execution times are listed. These were determined from the quoted minimum and maximum instruction execution times in the IBM 7090 Programmers Reference Manual.

The activity function program controls the execution of the individual activity functions. In this problem, 21 returns from each of the following six activity functions are calculated:

$$g_1(x) = x,$$

$$g_2(x) = x^2,$$

$$g_3(x) = \sqrt{x},$$

$$g_4(x) = 2 \sin x \, ,$$

$$g_5(x) = 2x \text{ if } 0 \leq x \leq 1 \, ,$$

$$= 4 - 2x \text{ if } 1 \leq x \leq 2 \, ,$$

$$g_6(x) = 2[x] \, .$$

The returns for each function are stored in a table. When the returns for all activity functions have been obtained, the maximization routine is executed. The maximization routine examines the return table for each activity function and a table of maximum returns for all previous activity functions. Then, given some resource, the returns from all resource combinations as applied to the current activity and to all previous activities are calculated. The greatest return is obtained and stored. The quantity of resource that generated this maximum return also is stored. When all quantities of resource have been tested against all activities, the result is a series of best policy tables.

The best policy tables contain for a given quantity of resource the portion of that resource that should be assigned to the respective activity. The remainder of the resource is then to be allotted to the remaining activities in the same manner.

The lookup routine has as an input the quantity of resource to be assigned to the given activities. The best policy table for the higher-order activity is examined. The entry in the table corresponding to the resource to be assigned is examined and a recommended assignment obtained. The remaining resource is then applied to the lower-order activities in the same manner. The result is a recommended assignment of resource to the activities that will generate a maximum return.

2. ACTIVITY FUNCTION RETURNS

a. General

The following are initialized: $x_{max}$, N, and $\Delta$, with $x_{max}$ the maximum

resource that can be allocated to any activity, N the number of activity functions, in this case six, and $\Delta$ the resource increment, in this case 0.1.

The activity counter, k, is set to 1 and the returns for resource allocations varying from 0 to 2 in steps of 0.1 are calculated. When the assigned resource, x, reaches the maximum resource available, $x_{max}$, the activity counter is increased by 1 and returns for the second activity are calculated.

When the activity counter has reached N = 6, the returns from all activities have been computed and stored in tables $g_k(x)$.

**b. Activity Function 1**

The return from activity function 1, $g_1(x) = x$, for a resource assignment of x is simply x.

**c. Activity Function 2**

The return from activity function 2, $g_2(x) = x^2$, for a resource assignment of x is the square of the assigned resource, $x^2$.

**d. Activity Function 3**

The return from activity function 3, $g_3(x) = \sqrt{x}$, for a resource assignment of x is the square root of the assigned resource, $\sqrt{x}$. The resource is represented in the computer as a floating point number. An initial guess for the square root is made,

$$G_k = \frac{1}{2}(f + 1),$$

where f is the fractional part of the floating point number. This guess is the input to the iterative portion of the routine.

After three iterations where successive approximations are obtained from

$$G_{k+1} = \frac{1}{2}\frac{f}{G_k} + G_k,$$

the exponent of the floating point number is tested to determine the exponent of the square root. Program control is then returned to the activity function program.

### e. Activity Function 4

The return from activity function 4, $g_4(x)$, for a resource assignment of x is 2 sine x. The input to the series is R, the residue of x mod $2\pi$. The sign of the sine is determined and quadrant correction of R is performed. Then

$$\alpha = \frac{2}{\pi} R$$

is calculated and used as input to the series; $\alpha$ and $\alpha^2$ are calculated and stored. The nested series approximation,

$$\alpha(C_1 + \alpha^2(C_3 + \alpha^2(C_5 + \alpha^2 C_7))) ,$$

is computed, the sign added, and the result stored. Program control is transferred back to the activity function program.

### f. Activity Function 5

The return from activity function 5, $g_5(x)$, for a resource assignment of x is 2x for $0 \leq x \leq 1$ and 4 - 2x for $1 \leq x \leq 2$.

### g. Activity Function 6

The return from activity function 6, $g_6(x) = 2[x]$, for a resource assignment of x is twice the largest integer equal to or less than x.

## 3. MAXIMIZATION

The activity counter k is initially set to 1. The $f_{k-1}(x)$ table is initially zeroed since the maximized returns from activity 0 are zero. The resource to be maximized, x, is initially assigned at 0. A storage location, $\beta$, containing intermediate maximum returns, is initially set at a maximum negative value to enable acceptance of the first return. $X_k$ is the

resource to be assigned, under the problem constraints, to activity k and to prior activities whose maximized returns for an assigned resource are the entries of the $f_{k-1}(x)$ table.

Index $i = x_k/\Delta$ is used to determine the return from an assignment of resource $x_k$ to activity k. Index $j = x - i/\Delta$ is used to determine the return from an assignment of resource $x - x_k$ to prior activities.

The sum of the returns, $\alpha$, is stored and compared with any previous return for the given resource, $\beta$. If the current return is larger than the previous maximum return, $\beta$ is replaced by $\alpha$ and the allocated resource that generated this return is stored in $\gamma$. If the current return is equal to or less than the previous maximum return, $\beta$ and $\gamma$ remain unchanged.

Then $x_k$ is incremented by $\Delta$ and if it is less than or equal to the current maximum resource to be tested, a new set of indices, i and j, are calculated and the above process repeated.

When $x_k$ is greater than x, indicating that all combinations of resource $x_k$ subject to the constraints have been used to determine the maximum return, then the value of the maximum return and the amount of resource that generates this return are stored in the $f_k(x)$ return table and $x_k(x)$ policy table, respectively.

Then x is increased by $\Delta$ and tested against $x_{max}$. If x is equal to or less than $x_{max}$, all combinations of x resource allocations consistent with the problem constraints are tested. If x is greater than $x_{max}$, then the $f_k(x)$ table is moved to the $f_{k-1}(x)$ table, and k is increased by 1 and tested. If k is equal to or less than N, then the maximizing process is repeated. If k is greater than N, the procedure stops.

At this time, return tables and policy tables are available in storage.

4. LOOKUP

In the lookup routine, k is initialized with N, the number of activities. The resource to be allocated is stored in x. The $x^{th}$ entry of the policy

table for the $k^{th}$ activity is obtained and stored in table θ. X is decreased by the amount that is assigned to activity k. The diminished x is then applied to the k - 1 activity and a recommended policy obtained. This process is repeated for the remaining activities. The end result is a table of recommended resource assignments to the k activities.

5.  CONCLUSIONS

The sequential execution of the sample dynamic programming problem takes 0.151 to 0.224 sec when programmed for an IBM 7090 (see Appendix III for Machine I parallel execution). Approximately 570 words of memory are used for program location and table storage.

6.  FLOW CHARTS AND PROGRAM TABLES

Figures II-1 through II-5 are the flow charts for the activity functions. Figure II-6 is the flow chart for maximation and Figure 7 is the lookup function flow chart.

Table II-1 shows the execution time for the dynamic programming technique on the IBM 7090. Tables II-2 through II-5 show, respectively, the programs for activity function control, maximization, the activity functions, and lookup. Table II-6 shows the common storage.
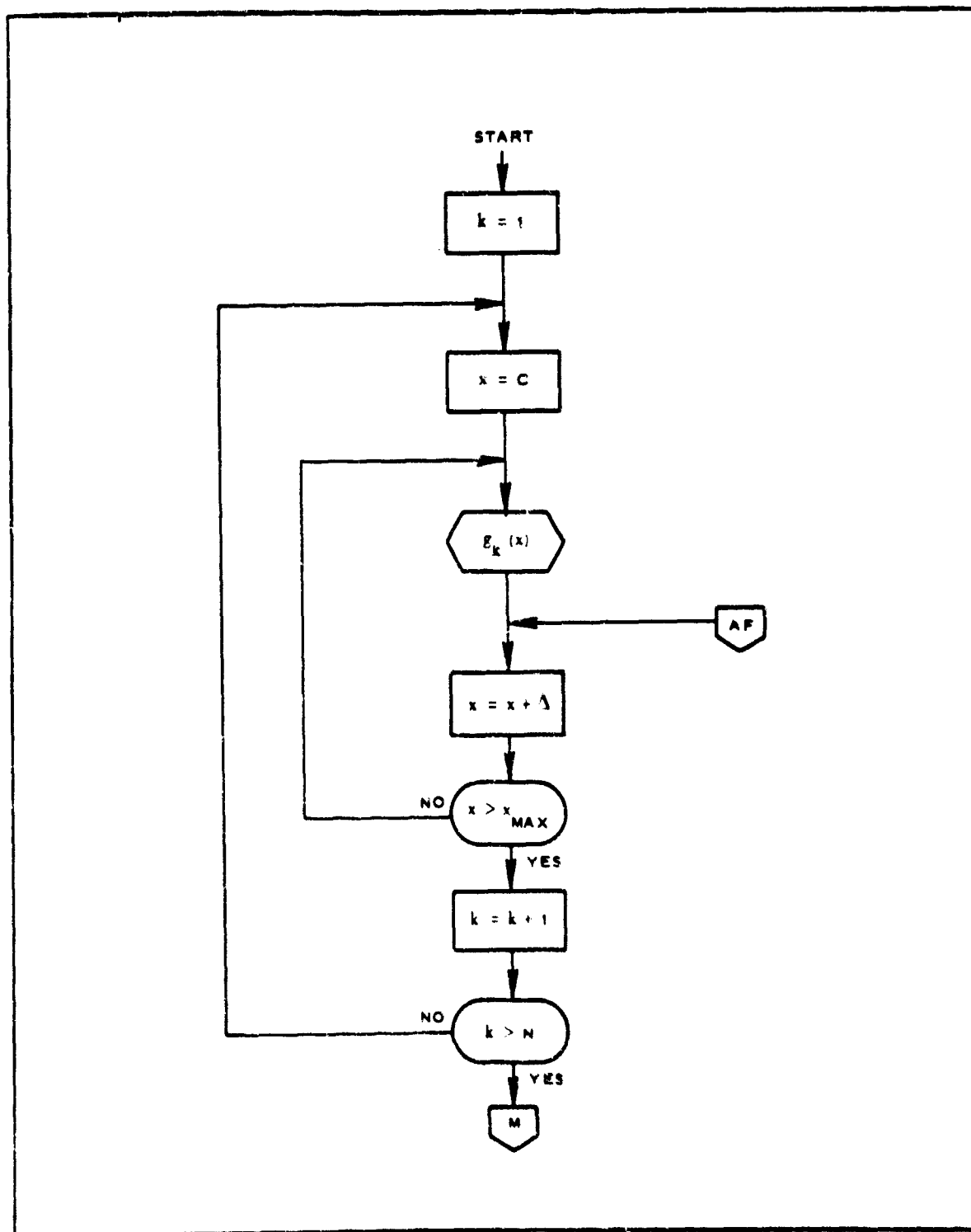
Figure II-1 - Activity Function Control Flow Chart, IBM 7090
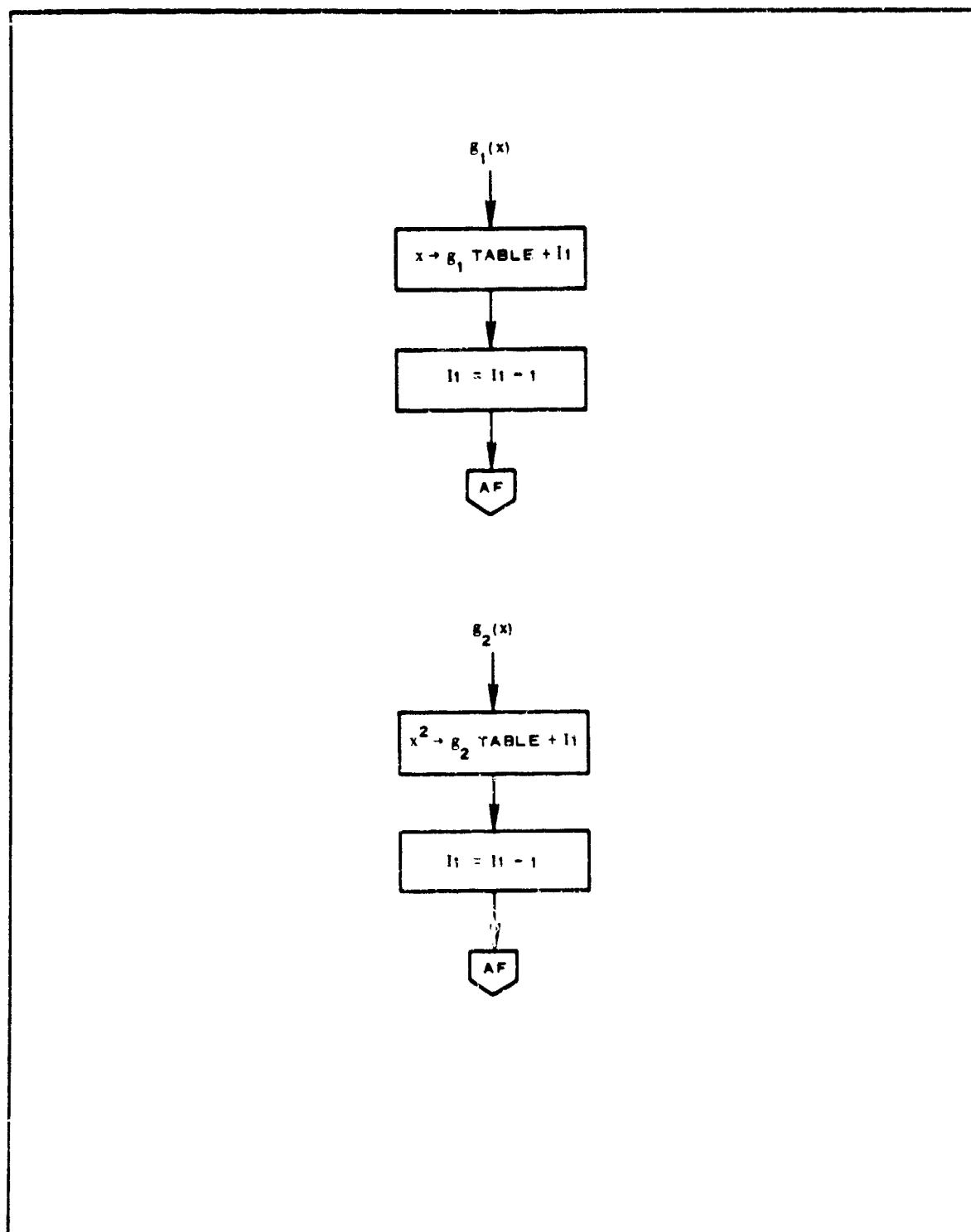
Figure II-2 - Activity Functions 1 and 2 Flow Chart, IBM 7090
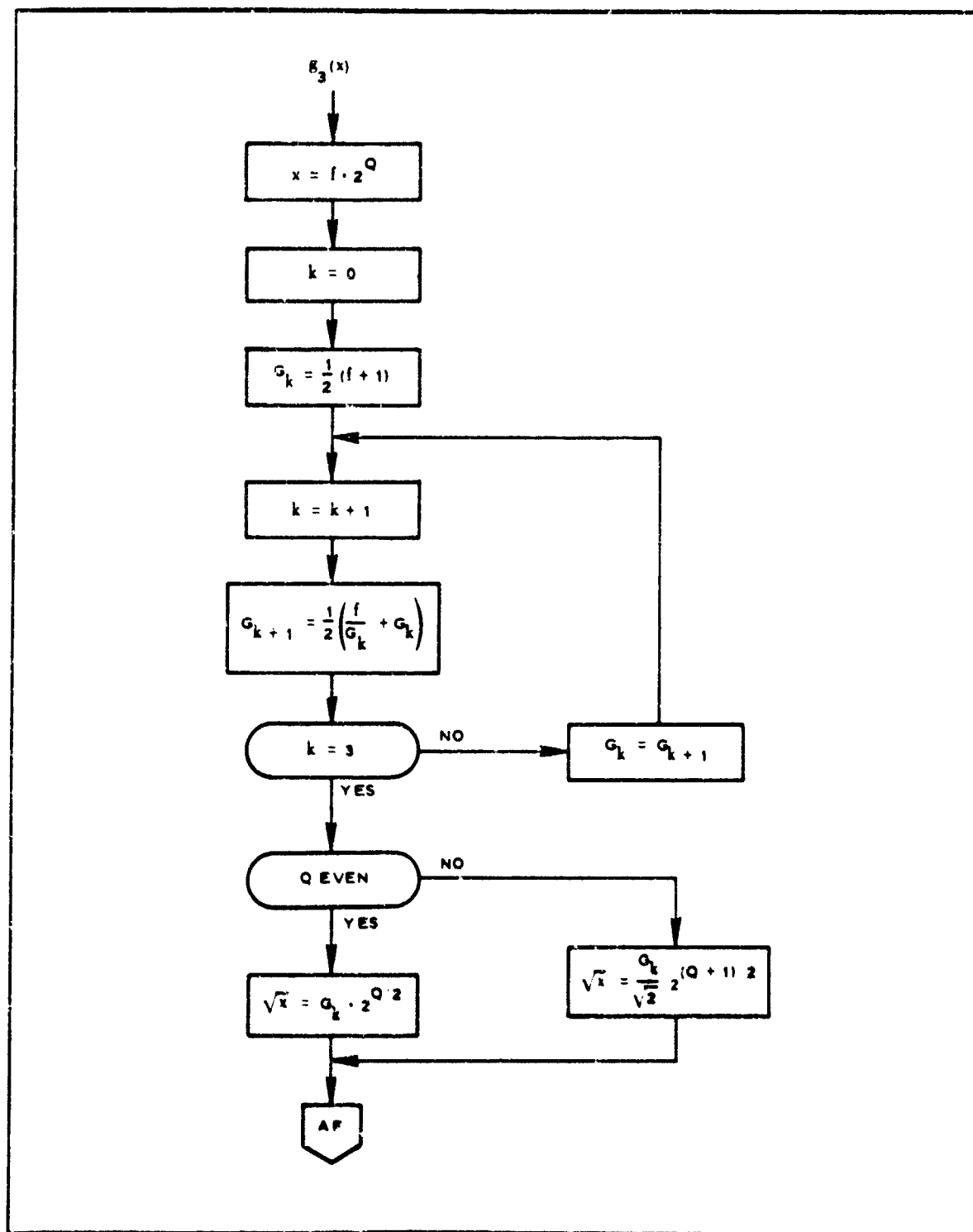
Figure II-3 - Activity Function 3 Flow Chart, IBM 7090

$B_4(x)$

$$R = |x|2\pi$$

$$k = 1 \text{ IF } R < \frac{\pi}{2}$$

$$k = -1 \text{ IF } R \geq \frac{\pi}{2}$$

R = R IF x IN Q1
R = $\pi$ - R IF x IN Q2
R = R - $\pi$ IF x IN Q3
R = 2$\pi$ - R IF x IN Q4

$$\alpha = \frac{2}{\pi} R$$

$$\text{SIN } x = k\left[C_1\alpha + C_3\alpha^3 + C_5\alpha^5 + C_7\alpha^7\right]$$

$$I_1 = I_1 - 1$$

AF
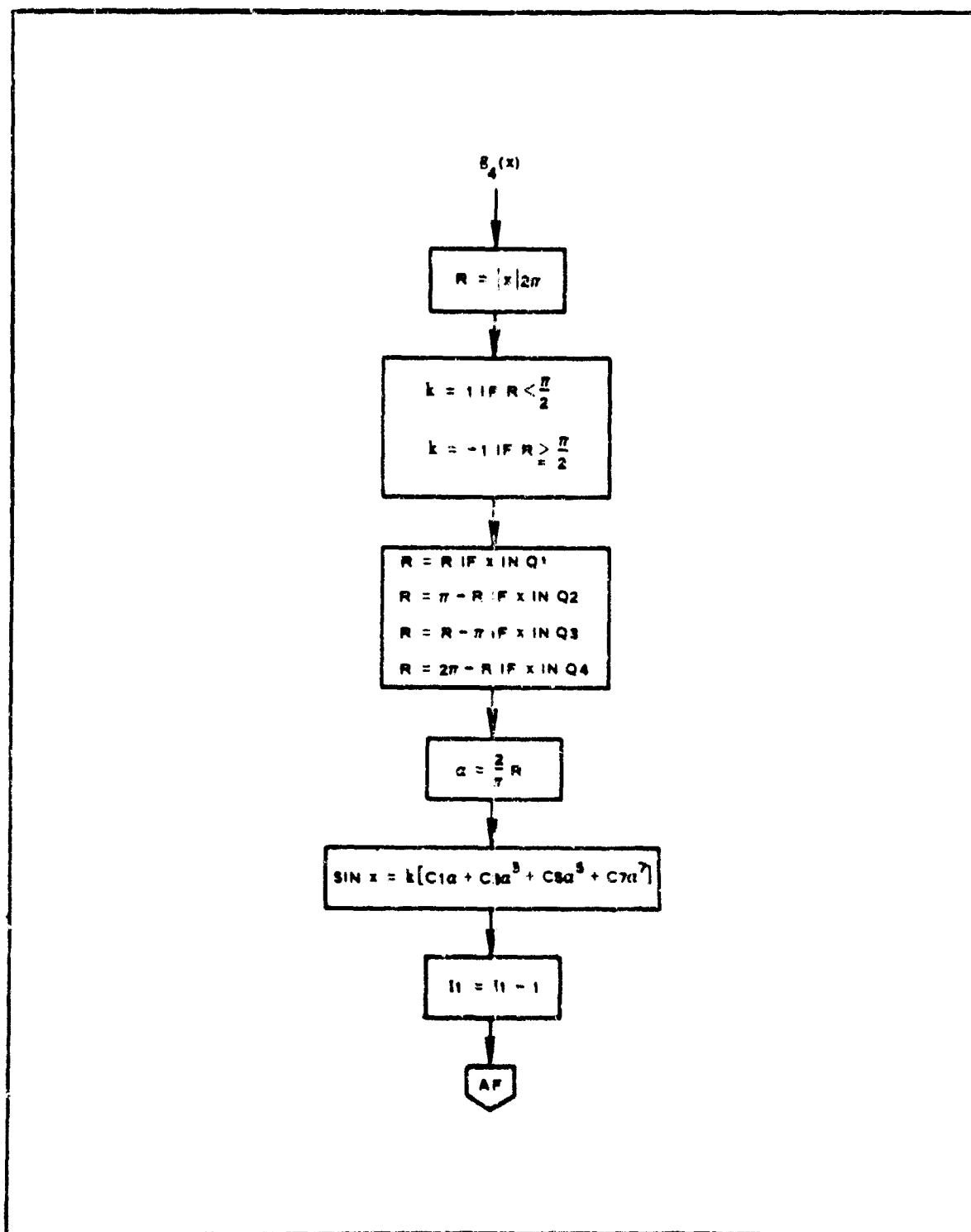
Figure II-4 - Activity Function 4 Flow Chart, IBM 7090
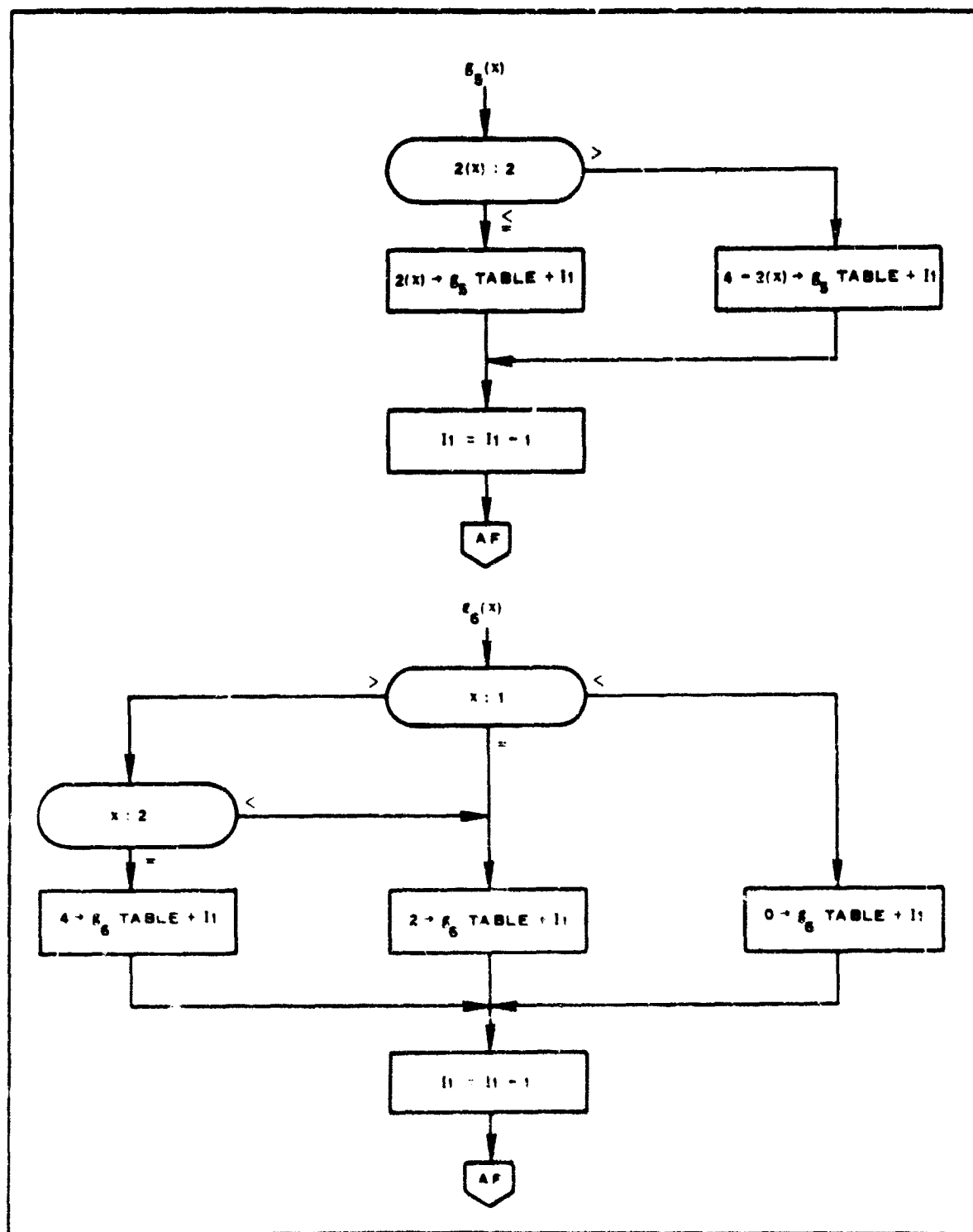
APPENDIX II



Figure II-5 - Activity Functions 5 and 6 Flow Chart, IBM 7090

-27-
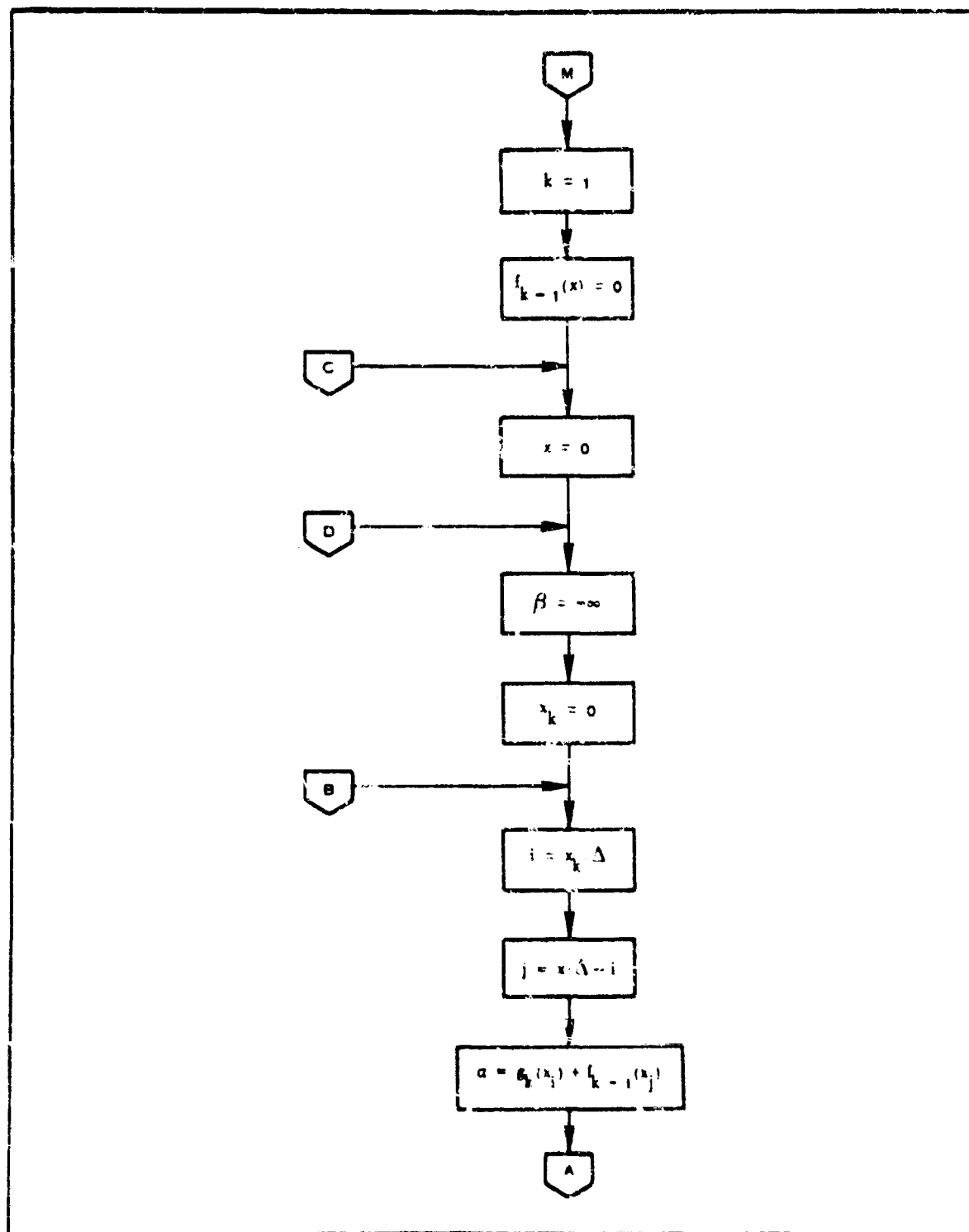
Figure II-6 - Maximization Function Flow Chart, IBM 7090 (Sheet 1)

Figure II-6 - Maximization Function Flow Chart, IBM 7090 (Sheet 2)
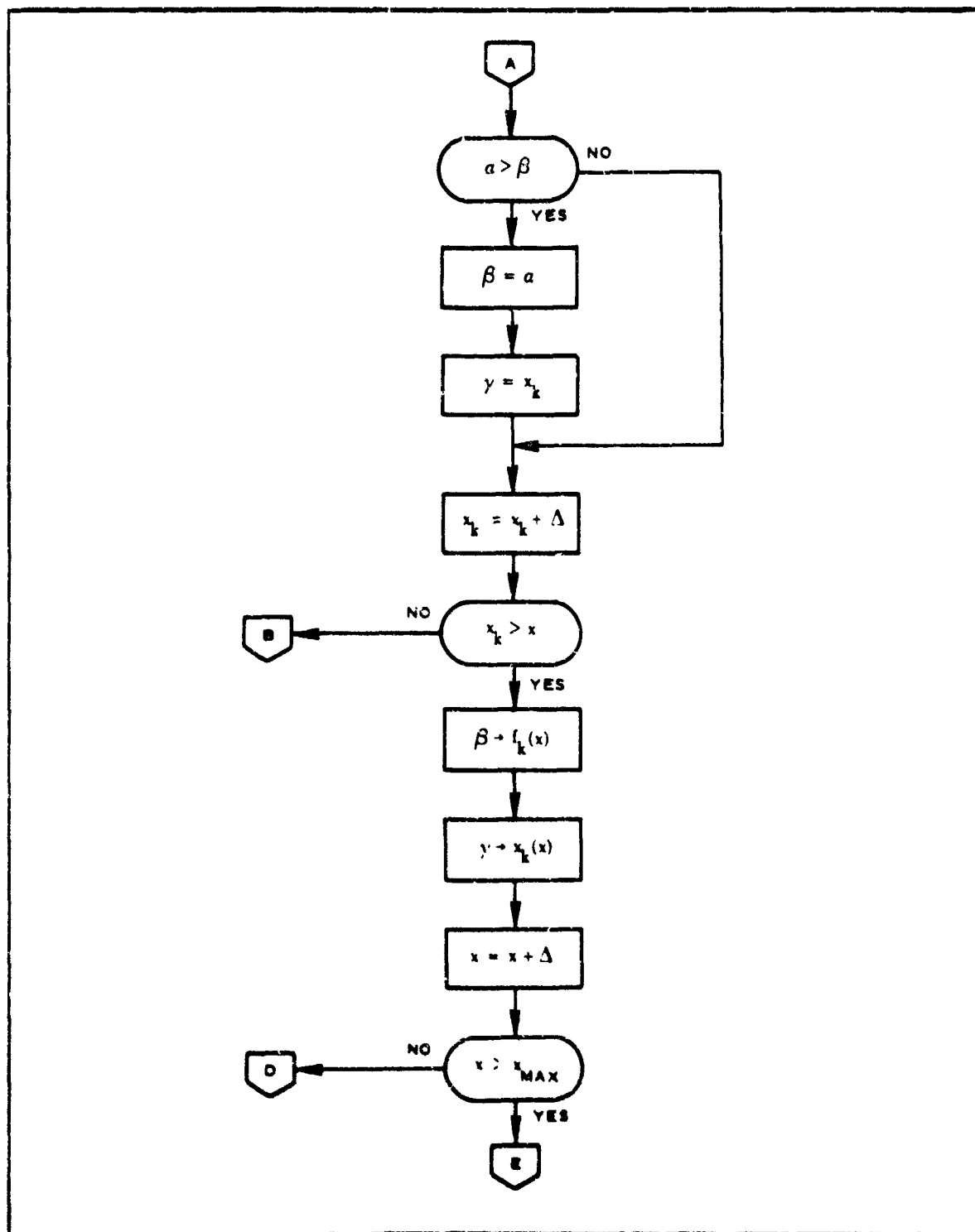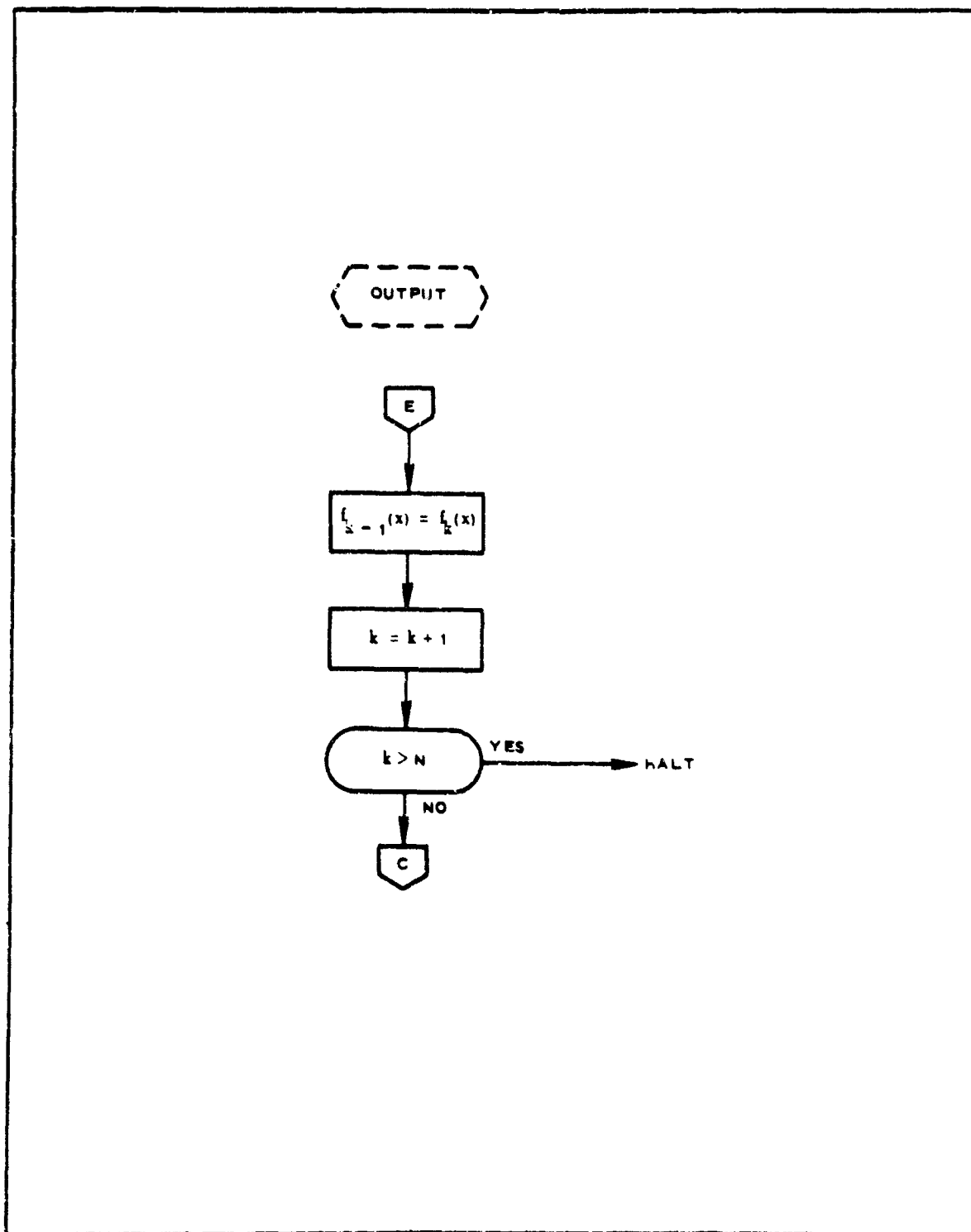
Figure II-6 - Maximisation Function Flow Chart, IBM 7090 (Sheet 3)
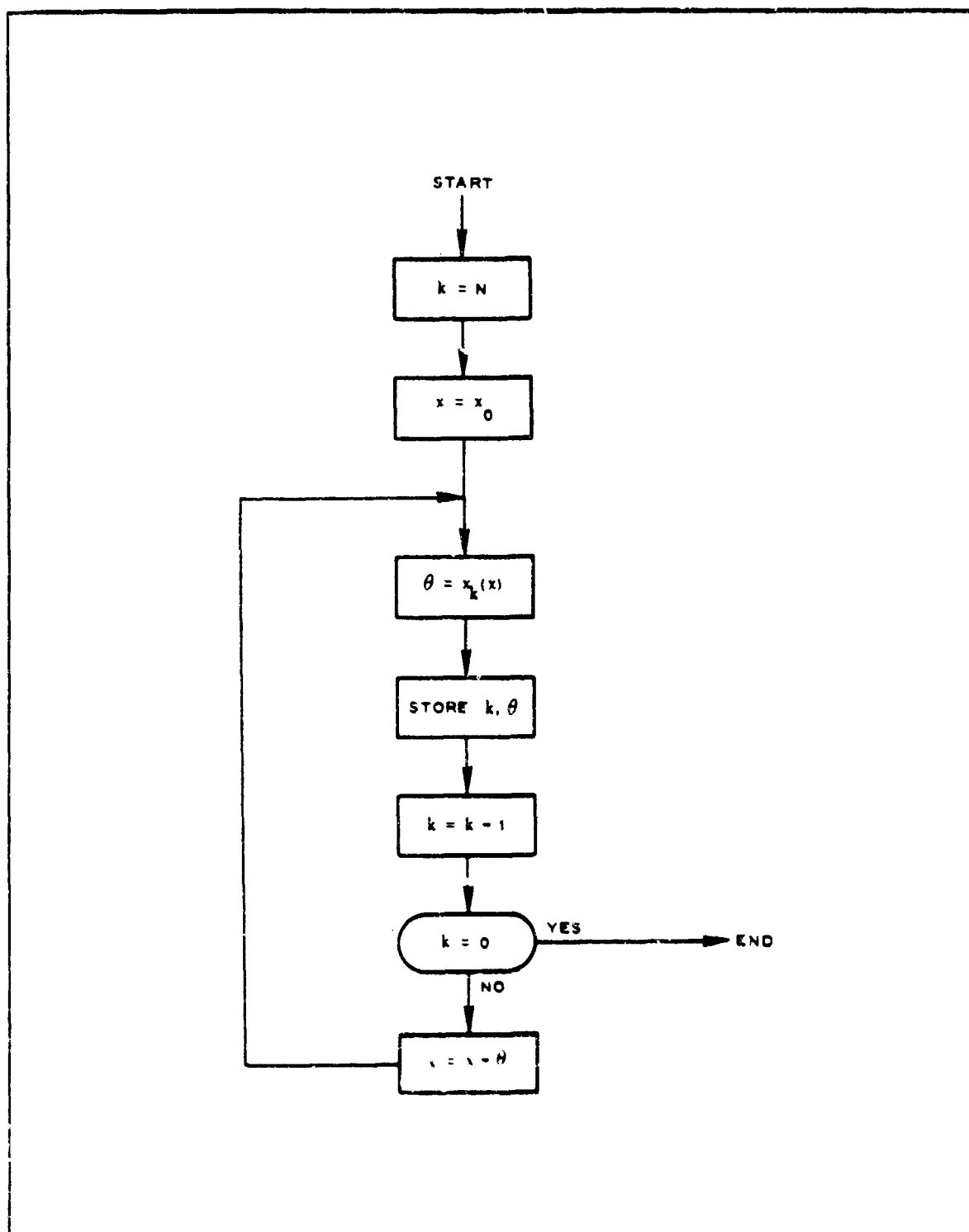
Figure II-7 - Lookup Function Flow Chart, IBM 7090

## TABLE II-1 · IBM 7090 EXECUTION TIME FOR

## DYNAMIC PROGRAMMING PROBLEM

| Item | Machine cycles | |
|---|---|---|
| | Minimum | Maximum |
| Activity function | 1,056 | 1,056 |
| $g_1(x)$ | 127 | 127 |
| $g_2(x)$ | 169 | 400 |
| $g_3(x)$ | 1,723 | 3,130 |
| $g_4(x)$ | 1,369 | 1,831 |
| $g_5(x)$ | 313 | 544 |
| $g_6(x)$ | 154 | 212 |
| Maximization | 64,398 | 95,430 |
| Lookup | 176 | 176 |
| Total machine cycles | 69,485 | 102,906 |
| Microseconds per machine cycle | 2.18 | 2.18 |
| Total microseconds | 151,477 | 224,335 |
| Total seconds | 0.151 | 0.224 |

## TABLE II-2 - ACTIVITY FUNCTION CONTROL

## PROGRAM, IBM 7090

| Item | Instruction | | | Remarks | Machine cycles |
|---|---|---|---|---|---|
| INIT | LXA | 2 | +SIX | $(T3) = \Delta$ | 2 |
| INIT6 | LXA | 1 | TWONE | $(T1) = 21_D$ | 2 |
| INIT2 | CLA | | T4 | $(T4) = c$ | 2 |
| INIT5 | STO | | x | $(x) = C$ or $x + \Delta$ | 2 |
| | TRA | 2 | ADDR | | 1 |
| INIT3 | CLA | | x | $x = x + \Delta$ | 2 |
| | ADD | | $\Delta$ | T3 | 2 |
| | CAS | | $x_{max}$ | | 3 |
| | TRA | | INIT4 | $>$ | 1 |
| | TRA | | INIT5 | $=$ | 1 |
| | TRA | | INIT5 | $<$ | 1 |
| INIT4 | TIX | 2,1 | INIT6 | 12    12 - 1 | 2 |

TABLE II-3 - MAXIMIZATION FUNCTION PROGRAM, IBM 7090

| Item | Instruction | | | Remarks | Machine cycles |
|------|------|------|------|---------|----------------|
| | CLA | | ONE | | 2 |
| | STO | | k | = 1 activity number | 2 |
| | LXA | 2 | TWONE | $(I2) = +21$ | 2 |
| RET1 | STZ | 2 | $f_{k-1}(x)$ TAB | $0 \rightarrow f_{k-1}(x)$ table | 2 |
| | TIX | 2, 1 | RET1 | | 2 |
| RET9 | STZ | | x | $0 \rightarrow$ x resource | 2 |
| RET2 | CLA | | $\div \infty$ | 77  77  77  77  77  77 | 2 |
| | STO | | $\beta$ | | 2 |
| | STZ | | $x_k$ | | 2 |
| RET5 | CLA | | $x_k$ | | 2 |
| | DVH | | $\Delta$ | $x^K/\Delta = i$ | 3 - 14 |
| | STQ | | IND1 | $= i$ | 2 |
| | CLA | | x | | 2 |
| | DVH | | $\Delta$ | $x/\Delta$ | 3 - 14 |
| | XCA | | | $(AC) = x/\Delta$ | 1 |
| | SUB | | IND1 | | 2 |
| | STO | | IND2 | $= j$ | 2 |
| | LXA | 2 | IND1 | $(I1) = k$ | 2 |
| | LXA | 4 | IND2 | $(I4) = j$ | 2 |
| | CLA | 2 | $g_k$ TAB | | 2 |
| | ADD | 4 | $f_{k-1}$ TAB | $(AC) = \alpha$ | 2 |
| | CAS | | $\beta$ | | 3 |
| | TRA | | RET3 | $>$ | 1 |
| | TRA | | RET4 | $=$ | 1 |
| | TRA | | RET4 | $<$ | 1 |
| RET3 | STO | | $\beta$ | $(\beta) = \alpha$ | 2 |
| | CLA | | $x_k$ | | 2 |
| | STO | | $\gamma$ | $(\gamma) = x_k$ | 2 |
| RET4 | CLA | | $x_k$ | | 2 |

-33-

TABLE II-3 - MAXIMIZATION FUNCTION PROGRAM, IBM 7090 (Continued)

| Item | Instruction | | | Remarks | Machine cycles |
|------|------|------|------|---------|----------------|
| | ADD | | $\Delta$ | | 2 |
| | STO | | $x_x$ | $x_k = x_k +$ | 2 |
| | CAS | | x | | 3 |
| | TRA | | RET6 | $>$ | 1 |
| | TRA | | RET5 | $=$ | 1 |
| | TRA | | RET5 | $<$ | 1 |
| RET6 | CLA | | $\beta$ | Best returns | 2 |
| | STO | 2 | $f_k(x)$ TAB | | 2 |
| | CLA | | $\gamma$ | | 2 |
| | STO | 2 | $x_k(x)$ TAB | | 2 |
| | CLA | | x | | 2 |
| | ADD | | $\Delta$ | | 2 |
| | STO | | x | $x = x + \Delta$ | 2 |
| | CAS | | $x_{max}$ | | 3 |
| | TRA | | OUT | $>$ | 1 |
| | TRA | | RET2 | $=$ | 1 |
| | TRA | | RET2 | $<$ | 1 |
| OUT | "OUTPUT" | | | | |
| | LXA | 2 | +TWONE | $=21$ | 2 |
| RET7 | CLA | 2 | $f_k(x)$ TAB | | 2 |
| | STA | 2 | $f_{k-1}(x)$ TAB | | 2 |
| | TIX | 2, 1 | RET7 | | 2 |
| | CLA | | k | | 2 |
| | ADD | | ONE | | 2 |
| | CAS | | N | | 3 |
| | TRA | | RET9 | $>$ | 1 |
| | TRA | | RET8 | $=$ | 1 |
| RET8 | STO | | k | $<$ | 2 |
| | TRA | | RET9 | | 1 |

## TABLE II-4 - ACTIVITY FUNCTIONS PROGRAM, IBM 7090

| Item | Instruction | | | Remarks | Machine cycles |
|---|---|---|---|---|---|
| $g_1(x)$ | CLA | | (x) | | 2 |
| | STO | 1 | G1TAB* | $x \longrightarrow$ G1TAB* + (I1) | 2 |
| | TIX | 1, 1 | INIT3 | | 2 |
| | TRA | | INIT3 | | 1 |
| $g_2(x)$ | LDQ | | (x) | | 2 |
| | FMP | | (x) | $x^2$ | 2 - 13 |
| | STO | 1 | G2TAB* | $x^2 \longrightarrow$ G2TAB* + (I1) | 2 |
| | TIX | 1, 1 | INIT3 | T1 = I1 - 1 | 2 |
| | TRA | | INIT3 | | 1 |
| $g_3(x)$ | SSP | | 3 | | 2 |
| | TZE | | C9-1 | | 2 |
| | STO | | COMMON | | 2 |
| | ANA | | C9 | | 3 |
| | LRS | | 1 | | 2 - 7 |
| | ADD | | COMMON | | 2 |
| | LRS | | 1 | | 2 - 7 |
| | ADD | | C10 | | 2 |
| | LXA | | $g_3(x)$ | | 2 |
| STO | STO | | COMMON + 1 | | 2 |
| | CLA | | COMMON | | 2 |
| | FDH | | COMMON + 1 | | 3 - 13 |
| | STQ | | COMMON + 2 | | 2 |
| | CLA | | COMMON + 2 | | 2 |
| | FAD | | COMMON + 1 | | 6 - 15 |
| | SUB | | C9 | | 2 |
| | TIX | 4, 1 | STO | | 2 |
| | TIX | 1, 1 | INIT3 | | 2 |
| | TRA | | INIT3 | | 1 |

## TABLE II-4 - ACTIVITY FUNCTIONS PROGRAM, IBM 7090 (Continued)

| Item | Instruction | | | | | | Remarks | Machine cycles |
|------|------|------|------|------|------|------|---------|---------------|
| C9 | OCT | | | 001 | 000 | 000 | 000 | |  |
| C10 | OCT | | | 100 | 400 | 000 | 000 | |  |
|  | SXA | 1 | TEM | | | | | 2 |
|  | SXA | 2 | TEM + 1 | | | | | 2 |
| $g_4(x)$ | CLA | | x | | | | | 2 |
| B | SUB | | $2\pi$ | | | | | 2 |
|  | CAS | | $2\pi$ | | | | | 3 |
|  | TRA | | B | | | | > | 1 |
|  | TRA | | B | | | | = | 1 |
|  | STO | | R | | | | $R < 2\pi$ | 2 |
|  | CAS | | TT | | | | | 3 |
|  | TXI | 1, 1 | E | | | | -, $I1 = 1$ | 2 |
| C | TXI | 1, 1 | E | | | | | 2 |
|  | CAS | | $\pi/2$ | | | | | 3 |
|  | TRA | | D | | | | $\pi/R$ | 1 |
|  | TRA | | D | | | | $\pi/R$ | 1 |
|  | TRA | | SER | | | | $R = R$ | 1 |
| E | CAS | | $3\pi/2$ | | | | | 3 |
|  | TRA | | G | | | | > | 1 |
|  | TRA | | G | | | | = | 1 |
| F | CLA | | R | | | | $R - \pi$ | 2 |
|  | SUB | | $\pi$ | | | | | 2 |
| H | STO | | R | | | | $R = R - \pi$ | 2 |
|  | TRA | | SER | | | | | 1 |
| G | CLA | | $2\pi$ | | | | | 2 |
| I | SUB | | R | | | | $2\pi - R$ | 2 |
|  | TRA | | H | | | | | 1 |
| D | CLA | | $\pi$ | | | | $\pi - R$ | 2 |

TABLE II-4 - ACTIVITY FUNCTIONS PROGRAM, IBM 7090 (Continued)

| Item | Instruction | | | Remarks | Machine cycles |
|------|------|------|------|---------|----------------|
| | TRA | | I | | 1 |
| SER | LDQ | | $2/\pi$ | | 2 |
| | FMP | | R | | 2 - 13 |
| | STO | | $\alpha$ | | 2 |
| | LDQ | | $\alpha$ | | 2 |
| | FMP | | $\alpha$ | | 2 - 13 |
| | STO | | $\alpha2$ | $\alpha2$ | 2 |
| | LXA | 2 | 3 | | 2 |
| | LDQ | 2 | C1 | | 2 |
| | FMP | | $\alpha2$ | | 2 - 13 |
| | TIX | 2, 1 | DE | | 2 |
| DE | FAD | 2 | C1 | | 6 - 15 |
| | STO | | TEM | | 2 |
| | LDQ | | TEM | | 2 |
| | TIX | 2, 0 | DF | | 2 |
| | FMP | | $\alpha$ | | 2 - 13 |
| | STO | | SIN (x') | | 2 |
| | TIX | 1, 0 | SN | | 2 |
| | TRA | | SN + 2 | | 1 |
| SN | CLA | | -0 | | 2 |
| | SBN | | SiN (x') | | 2 |
| SN + 2 | LXA | 1 | TEM | | 2 |
| | LXA | 2 | TEM + 1 | | 1 |
| | STO | 1 | G4TAB* | | 2 |
| | TIX | 1, 1 | INIT3 | | 2 |
| | TRA | | INIT3 | | 1 |
| $g_5(x)$ | LDQ | | x | | 2 |
| | FMP | | TWO | | 2 - 13 |

TABLE II-4 - ACTIVITY FUNCTIONS PROGRAM, IBM 7090 (Continued)

| Item | Instruction | | | Remarks | Machine cycles |
|------|------|------|------|---------|----------------|
| | CAS | | TWO | | 3 |
| | TRA | | A2 | > | 1 |
| | TRA | | A1 | = | 1 |
| A1 | STO | 1 | G5TAB* | < | 2 |
| | TIX | 1, 1 | INIT3 | | 2 |
| | TRA | | INIT3 | | 1 |
| A2 | STO | | TEM | | 2 |
| | CLA | | FOUR | | 2 |
| | SUB | | TEM | | 2 |
| | TRA | | A1 | | 1 |
| $g_6(x)$ | CLA | | x | | 2 |
| | CAS | | ONE | | 3 |
| | TRA | | B1 | > | 1 |
| | TRA | | B3 | = | 1 |
| | STZ | 1 | G6TAB* | < | 2 |
| B4 | TIX | 1 | INIT3 | | 2 |
| | TRA | | INIT3 | | 1 |
| B1 | CAS | | TWO | | 3 |
| | HLT | | | > | |
| | TRA | | B2 | = | 1 |
| B3 | CLA | | TWO | < | 2 |
| B5 | STO | 1 | G6TAB* | | 2 |
| | TRA | | B4 | | 1 |
| B2 | CLA | | FOUR | | 2 |
| | TRA | | B5 | | 1 |

TABLE II-5 - LOOKUP FUNCTION PROGRAM, IBM 7090

| Item | Instruction | | | Remarks | Machine cycles |
|---|---|---|---|---|---|
| | CLA | | N | | 2 |
| | STO | | k | Highest activity number | 2 |
| | CLA | | x | | 2 |
| | STO | | x | | 2 |
| LOK3 | LXA | 2 | k | | 2 |
| | CLA | 2 | $ADDRX_k(x)$ | | 2 |
| | STA | | LOK1 | | 2 |
| | LXA | 4 | x | | 2 |
| LOK1 | CLA | 4 | $x_k(x)TAB$ | | 2 |
| | STO | 2 | $\theta TAB$ | Best allocation for $k^{th}$ activity | 2 |
| | CLA | | k | | 2 |
| | SUB | | 1 | | 3 |
| | CAS | | ZERO | | 1 |
| | TRA | | LOK2 | > | 1 |
| | HLT | | END | = | 2 |
| | HLT | | | < | 2 |
| LOK2 | STO | | k | | 2 |
| | CLA | | x | | 2 |
| | SUB | 2 | $\theta TAB$ | | 2 |
| | STO | | x | | 1 |
| | TRA | | LOK3 | | |

## TABLE II-6 - COMMON STORAGE

| Item | Data | | Remarks |
|---|---|---|---|
| T4 | C | | |
| T3 | $\Delta$ | | |
| T2 | N | | |
| T1 | $x_{max}$ | | |
| +6 | TRA | $g_1(x)$ | |
| +5 | TRA | $g_2(x)$ | |
| +4 | TRA | $g_3(x)$ | |
| +3 | TRA | $g_4(x)$ | |
| +2 | TRA | $g_5(x)$ | |
| +1 | TRA | $g_6(x)$ | |
| ADDR | | | |
| SIX | | | |
| TWONE | | | |
| ONE | | | |
| k | | | |
| $-\infty$ | | | |
| $\beta$ | | | |
| $x_k$ | | | |
| IND1 | | | |
| IND2 | | | |
| ADDR $x_k(x)$ | | | |
| | ADDRESS | $x_1(x)$TAB | |
| | ADDRESS | $x_2(x)$TAB | |
| | ADDRESS | $x_3(x)$TAB | |
| | ADDRESS | $x_4(x)$TAB | |
| | ADDRESS | $x_5(x)$TAB | |
| | ADDRESS | $x_6(x)$TAB | |
| X0 | | | |

TABLE II-6 - COMMON STORAGE (Continued)

| Item | Data | Remarks |
|------|------|---------|
| Y | | |
| ZERO | | |
| COMMON | | |
| COMMON + 1 | | |
| COMMON + 2 | | |
| COMMON + 3 | | |
| C9 | | |
| C10 | | |
| TEM | | |
| TEM + 1 | | |
| C7 | 400.002310715 | -0.00467377 |
| C5 | 000.050632127 | 0.07968968 |
| C3 | 400.512567405 | -0.64596371 |
| C1 | 001.444176646 | 1.57079630 |
| $2/\pi$ | 000.505746037 | 0.6366198 |
| 3 | | |
| R | | |
| $2\pi$ | 006.220773230 | 6.2831853 |
| $3\pi/2$ | 004.554574363 | 4.7123889 |
| $\pi$ | 003.110325514 | 3.1415927 |
| $\pi/2$ | 001.444176646 | 1.5707963 |
| SINX' | | |
| -0 | | |
| $\alpha$ | | |
| $\alpha2$ | | |
| TWO | | |
| FOUR | | |

Figure II-8 - Table Layout, IBM 7090

# APPENDIX III - PROGRAMMING OF THE DYNAMIC PROGRAMMING

## TECHNIQUE FOR MACHINE I (PARALLEL)

## 1. INTRODUCTION

Described in this appendix is the sample problem that was programmed
for Machine I using the dynamic programming technique. The objectives
were to develop parallel solutions and programming techniques and to
determine what difficulties might arise in programming for Machine I.
Hence, the sample problem was kept small and no attempt was made to
extract maximum parallelism and speed.

The narratives and programs for the Machine I sample-problem program-
ming are detailed under Item 2; this description revolves around six ac-
tivity functions, a maximization function, and a lookup function. The
Machine I programming results are presented under Item 3. Under
Item 4, these results are compared with those resulting from the pro-
gramming of the same problem for a sequential computer.

## 2. NARRATIVES AND PROGRAMS

### a. Activity Function 1

The flow chart for activity function 1, $g_1(x) = x$, is shown in Figure III-1;
the program, in Table III-1; and the data vector format, in Table III-2.

This function is started by storing an ENB1 $p_1$, JMP G1 instruction in
the WAIT LIST. A free processor takes this instruction, enters $g_1$ into
index register 1, and jumps to G1, which is the address of the first in-
struction of the $g_1(x)$ activity function.

Index register 2 is initialized with zero, and index register 3 is loaded
with n, which is the number of times $g_1(x)$ is to be calculated. The

START

INITIALIZE
$\beta_1$, j, n

$x \rightarrow \beta_1 + j + 3$

$x = x + \Delta$

$u_1, 0, j \rightarrow u_1$ MAX

$j = j + 1$ ← NO ← $j = n$ → YES

HALT
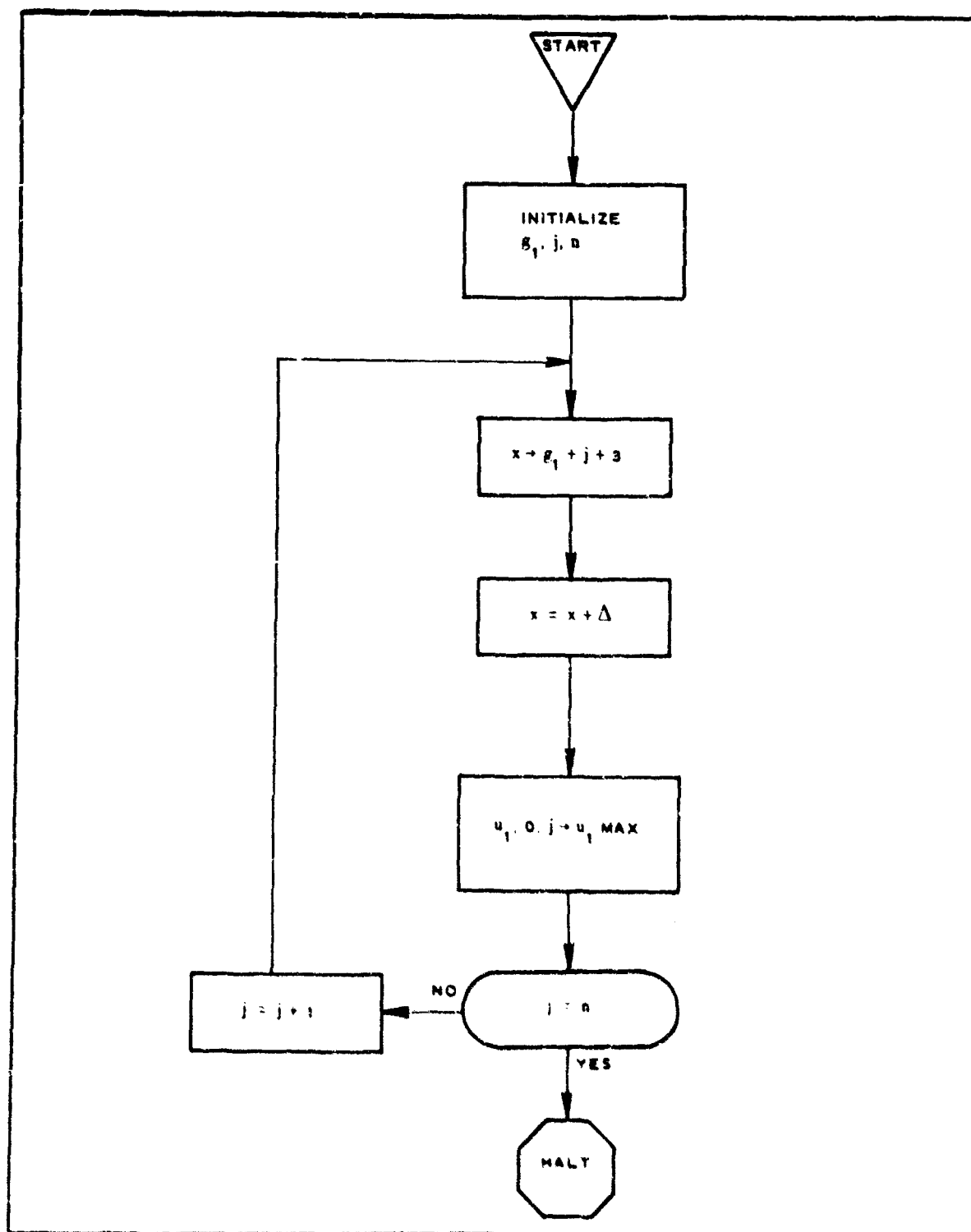
Figure III-1 - Activity Function 1 Flo      art, Machine I

## TABLE III-1 - ACTIVITY FUNCTION 1 PROGRAM,

## MACHINE I

| Item | Instruction | | | Remarks | Time ($\mu$sec) |
|---|---|---|---|---|---|
| | ENB1 | | $g_1$ | | 30 |
| | JMP | | G1 | | |
| G1 | ENB2 | | $\emptyset$ | | 30 |
| | LDB3 | 1 | 2 | n | |
| | LDA | 1 | $\emptyset$ | | 30 |
| | NOP | | | | |
| G1 + 2 | STA | 1 2 | 3 | | 30 |
| | ADD | 1 | 1 | | |
| | LRS | | 72 | | 30 |
| | LDA | | GIM | | |
| | INAL | 2 | 3 | | 30 |
| | STA | 2 | MTEM | | |
| | LDA | | GIM + 1 | LDI___, JMP MAX | 30 |
| | BGN | 2 | MTEM | | |
| | LLS | | 72 | | 30 |
| | ISK2 | 3 | $\emptyset$ | | |
| | JMP | | G1 + 2 | | 30 |
| | HLT | | | | |
| GIM | u | | 0 | | |
| GIM + 1 | 0 | | j | | |
| | LDI | | | | |
| | JMP | | MAX | | |

## TABLE III-2 - ACTIVITY FUNCTIONS 1 AND 2 DATA VECTOR

### FORMATS, MACHINE I

| Function $g_1(x)$ | | Function $g_2(x)$ | | | |
|---|---|---|---|---|---|
| Address | Data | Address | Data | Address | Data |
| $g_1$ | x | $g_2$ | x | $g_2 + 24$ | TEMP x |
| $g_1 + 1$ | $\Delta$ | $g_2 + 1$ | $\Delta$ | $g_2 + 25$ | TEMP x + 1$\Delta$ |
| $g_1 + 2$ | n | $g_2 + 2$ | n | $g_2 + 26$ | TEMP x + 2$\Delta$ |
| $g_1 + 3$ | $x_0$ | $g_2 + 3$ | $x_0^2$ | $g_2 + 27$ | TEMP x + 3$\Delta$ |
| $g_1 + 4$ | $x_1$ | $g_2 + 4$ | $x_1^2$ | $g_2 + 28$ | TEMP x + 4$\Delta$ |
| $g_1 + 5$ | $x_2$ | $g_2 + 5$ | $x_2^2$ | $g_2 + 29$ | TEMP x + 5$\Delta$ |
| $g_1 + 6$ | $x_3$ | $g_2 + 6$ | $x_3^2$ | $g_2 + 30$ | TEMP x + 6$\Delta$ |
| $g_1 + 7$ | $x_4$ | $g_2 + 7$ | $x_4^2$ | $g_2 + 31$ | TEMP x + 7$\Delta$ |
| $g_1 + 8$ | $x_5$ | $g_2 + 8$ | $x_5^2$ | $g_2 + 32$ | TEMP x + 8$\Delta$ |
| $g_1 + 9$ | $x_6$ | $g_2 + 9$ | $x_6^2$ | $g_2 + 33$ | TEMP x + 9$\Delta$ |
| $g_1 + 10$ | $x_7$ | $g_2 + 10$ | $x_7^2$ | $g_2 + 34$ | TEMP x + 10$\Delta$ |
| $g_1 + 11$ | $x_8$ | $g_2 + 11$ | $x_8^2$ | $g_2 + 35$ | TEMP x + 11$\Delta$ |
| $g_1 + 12$ | $x_9$ | $g_2 + 12$ | $x_9^2$ | $g_2 + 36$ | TEMP x + 12$\Delta$ |
| $g_1 + 13$ | $x_{10}$ | $g_2 + 13$ | $x_{10}^2$ | $g_2 + 37$ | TEMP x + 13$\Delta$ |
| $g_1 + 14$ | $x_{11}$ | $g_2 + 14$ | $x_{11}^2$ | $g_2 + 38$ | TEMP x + 14$\Delta$ |
| $g_1 + 15$ | $x_{12}$ | $g_2 + 15$ | $x_{12}^2$ | $g_2 + 39$ | TEMP x + 15$\Delta$ |
| $g_1 + 16$ | $x_{13}$ | $g_2 + 16$ | $x_{13}^2$ | $g_2 + 40$ | TEMP x + 16$\Delta$ |
| $g_1 + 17$ | $x_{14}$ | $g_2 + 17$ | $x_{14}^2$ | $g_2 + 41$ | TEMP x + 17$\Delta$ |
| $g_1 + 18$ | $x_{15}$ | $g_2 + 18$ | $x_{15}^2$ | $g_2 + 42$ | TEMP x + 18$\Delta$ |
| $g_1 + 19$ | $x_{16}$ | $g_2 + 19$ | $x_{16}^2$ | $g_2 + 43$ | TEMP x + 19$\Delta$ |
| $g_1 + 20$ | $x_{17}$ | $g_2 + 20$ | $x_{17}^2$ | $g_2 + 44$ | TEMP x + 20$\Delta$ |
| $g_1 + 21$ | $x_{18}$ | $g_2 + 21$ | $x_{18}^2$ | | |
| $g_1 + 22$ | $x_{19}$ | $g_2 + 22$ | $x_{19}^2$ | | |
| $g_1 + 23$ | $x_{20}$ | $g_2 + 23$ | $x_{20}^2$ | | |

contents of the first location of the $g_1$ vector, which is x, is then stored in the third word of the vector. Here x is the initial value of the vector from which all subsequent values of the activity function are calculated; it is increased by $\Delta$, then shifted into Q, and a series of instructions is executed to enable the starting of processors to operate on the maximization routine.

The maximization routine needs the beginning address of the vector and the value of the resource for which activity function returns are currently available. For every return calculated for the $g_1(x)$ activity function, a processor is started. The information transferred to this processor is $u_1$, o, and j.

Inserted into the address field of the instruction at GIM + 1 is the location where the contents of the index registers are stored. The instruction is then stored in the WAIT LIST for an available processor. The contents of Q are now shifted back into A. Index 2 is tested to determine if all iterations have been completed. If not completed, index 2 is incremented, and the loop is repeated. When index 2 equals 21, the operation is halted.

Successive values of the function are stored in successive locations of the vector. Each location has a unique name as determined by B1 + B2 + 3; B1 equals $g_1$; B2 is incremented by 1 for each iteration. Successive names of elements of the vector are $g_1 + 0 + 3$, $g_1 + 1 + 3$, $g_1 + 2 + 3 + \ldots$.

b. Activity Function 2

The flow chart for the $g_2(x) = x^2$ activity function is shown in Figure III-2; the program in Table III-3, and the data vector format, in Table III-2 along with the $g_1(x)$ format.

This function is started in the same manner as activity function $g_1(x)$. When there is an available processor and the JMP G2 is executed, zero is
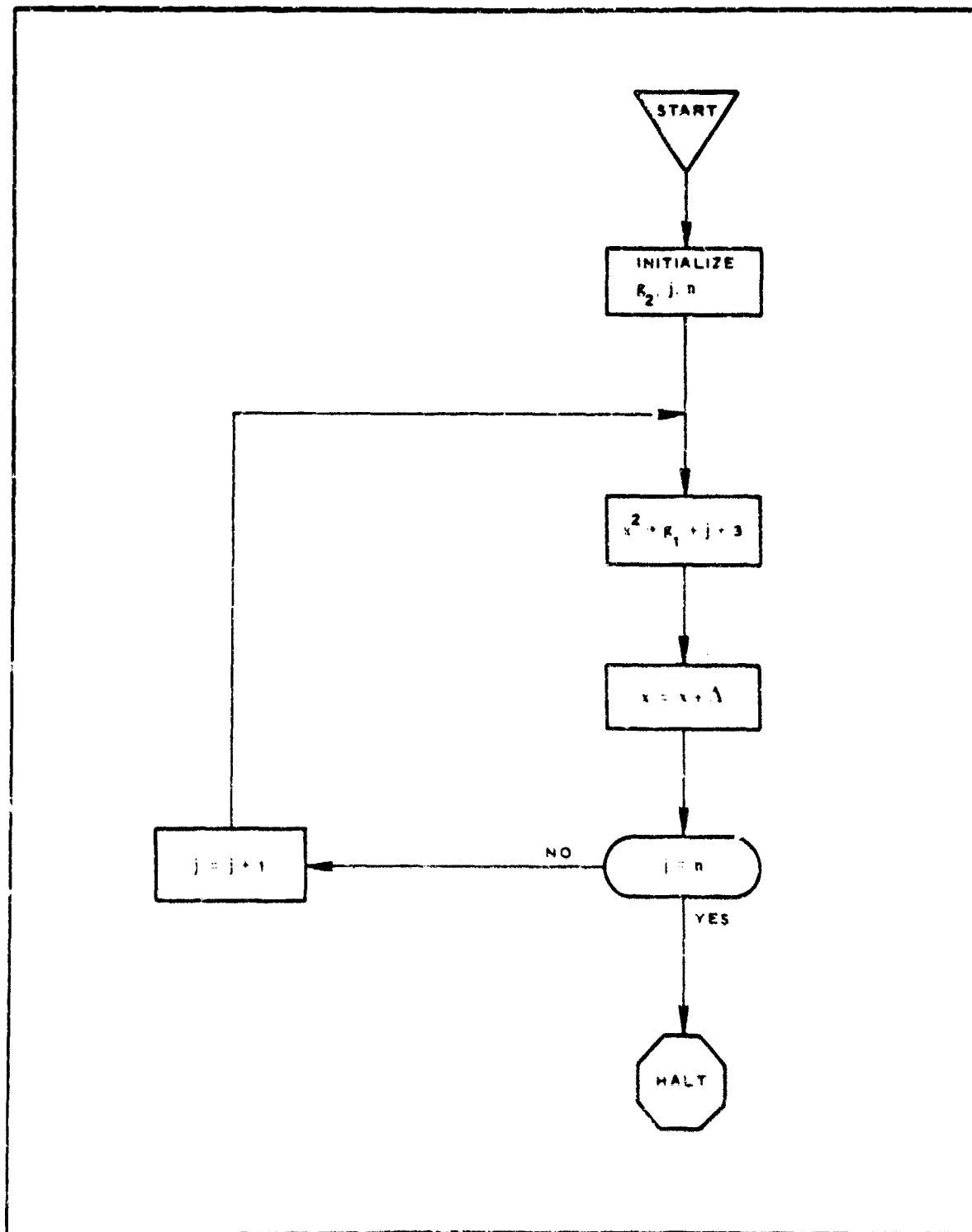
Figure III-2 - Activity Function 2 Flow Chart, Machine I

TABLE III-3 - ACTIVITY FUNCTION 2 PROGRAM.

MACHINE I

| Item | Instruction | | | | | Remarks | Time (µsec) |
|------|------|---|---|---|------|---------|-------------|
|      | ENB1 |   |   |   | $g_2$ |         | 30 |
|      | JMP  |   |   |   | G2   |         |    |
| G2   | ENB2 |   |   |   | $\emptyset$ |  | 30 |
|      | LDB3 | 1 |   |   | 2    | n       |    |
|      | LDA  | 1 |   |   | $\emptyset$ |  | 30 |
|      | NOP  |   |   |   |      |         |    |
| G2B  | STA  | 1 | 2 | 3 | 3    |         | 60 |
|      | FMP  | 1 | 2 | 3 | 3    |         |    |
|      | STA  | 1 | 2 |   | 3    |         | 30 |
|      | SEH  | 1 | 2 | 3 | 3    |         |    |
|      | LDA  | 1 | 2 | 3 | 3    |         | 30 |
|      | FAD  | 1 |   |   | 1    |         |    |
|      | NOP  |   |   |   |      |         | 30 |
|      | ISK2 |   |   | 3 | $\emptyset$ |  |    |
|      | JMP  |   |   |   | G2B  |         | 30 |
|      | HLT  |   |   |   |      |         |    |

entered into index 2, and $B1 + 2 = n$ is entered into index 3. The initial value of the data vector, x, is stored temporarily in an address equal to $B1 + B2 + B3 + 3$. On the first execution of the routine, this address is equal to $g_2 + 0 + n + 3$. In subsequent executions. B2 is incremented up to a maximum of $n = B3$. Hence, for each execution there is a unique temporary storage address where $x_i$ is stored prior to forming $x_i^2$. Then $x_i^2$ is stored in the data vector at the address equal to $B1 + B2 + 3$. Now $x_i$ is erased, and this location becomes an available word of memory capable of being named and used by another routine. This routine is re-peated n times until all values of the activity function have been calculated.

Activity functions $g_1(x)$ and $g_2(x)$ are quite simple. Except for the method of starting the processors, they could be run on a sequential machine.

### c. Activity Function 3

The flow chart for the $g_3(x) = \sqrt{x}$ activity function is shown in Figure III-3; the program, in Table III-4; and the data vector format in Table III-5.

This function is started by transferring indices $g_3$, 2, and 0 to an available processor and jumping to the subprogram, INIT. Indices $g_3$, 2, and 0 are in the $\beta 1$ table, and the LDI $\beta 1$, JMP I instruction is stored in the WAIT LIST to start an available processor.

The term, $\beta 1$, designates index values that are used as inputs to the I subprogram; and $\beta 2$ designate index values used as inputs to the LOOP and Q subprograms. Each time one of these indices is stored, an instruction is also stored in the WAIT LIST. This instruction has the pertinent $\beta$ and subprogram address to enable the processor to acquire the index values and to jump to the appropriate subprogram. Since the index transfer operation is complete at the time of the jump to the subprogram, and since the index value in the $\beta$ table is no longer of use, this information is erased from the $\beta$ table.

Three more processors are started with indices $g_3$, 0, and 0; $g_3$, 1, 0; $g_3$, 2, 0. These indices are stored in $\beta 2$ and are used to start the subprogram LOOP. Each time a processor is started on LOOP, another one is started on subprogram Q.

The $g_3$, 2, and 0 indices sent to the I program were the beginning of a tree of indices generated to permit parallel calculation of the square roots. With an index of $i = 2$ as input to the I program, $2i$ and $2i - 1$ are generated and used as inputs to the I program and to the LOOP program; $2i$ and $2i - 1$ in turn generate $4i$, $4i - 1$, $4i - 2$, and $4i - 3$. Eventually, a calculated index exceeds the vector size and index calculation halts.

The LOOP subprogram calculates the square root using Newton's iteration. The Q program determines whether the exponent is even or odd, to determine the exponent of the square root.

In addition to the data vector generated by the program, three other

$\beta_2 = g_3, 0, 0$

$= g_3, 1, 0$

$= g_3, 2, 0$

$= g_3, i, 0$

$\beta_1 = g_3, 2, 0$

START

$i = 2i - 1$

$i > n$ — NO → $g_3, i, 0 \rightarrow \beta_1, \beta_2$

$i = i + 1$

$i > n$ — NO → $g_3, i, 0 \rightarrow \beta_1, \beta_2$

YES

HALT

STA

$N = x$

$g_i = \dfrac{f}{2}$

$g_i + 1 = \dfrac{}{2}$

NO — ITNC

$g_i \rightarrow g_3$

HA

A

$\beta 2 = \beta_3, 0, 0$

$= \beta_3, 1, 0$

$= \beta_3, 2, 0$

$= \beta_3, i, 0$

START

$N = x + i\Delta$

$\beta_i = \frac{f}{2} + \frac{1}{2}$

$\beta_i + 1 = \frac{1}{2}\left(\frac{f_i}{\beta_i} + \beta_i\right)$

NO

ITNO = 3

YES

$\beta_i \rightarrow \beta_3 + i + T1$

HALT

$\beta 3 = \beta_3, \cap, 0$

$= \beta_3, 1, 0$

$= \beta_3, 2, 0$

$= \beta_3, i, 0$

START

$Q_i$ ODD

YES

·IO

$\frac{Q}{2} + \beta_i = \sqrt{N}$

$\frac{Q+1}{2} + \frac{\beta_i}{\sqrt{2}} = \sqrt{N}$

$\sqrt{N} \rightarrow \beta_3 + L + 3$

HALT
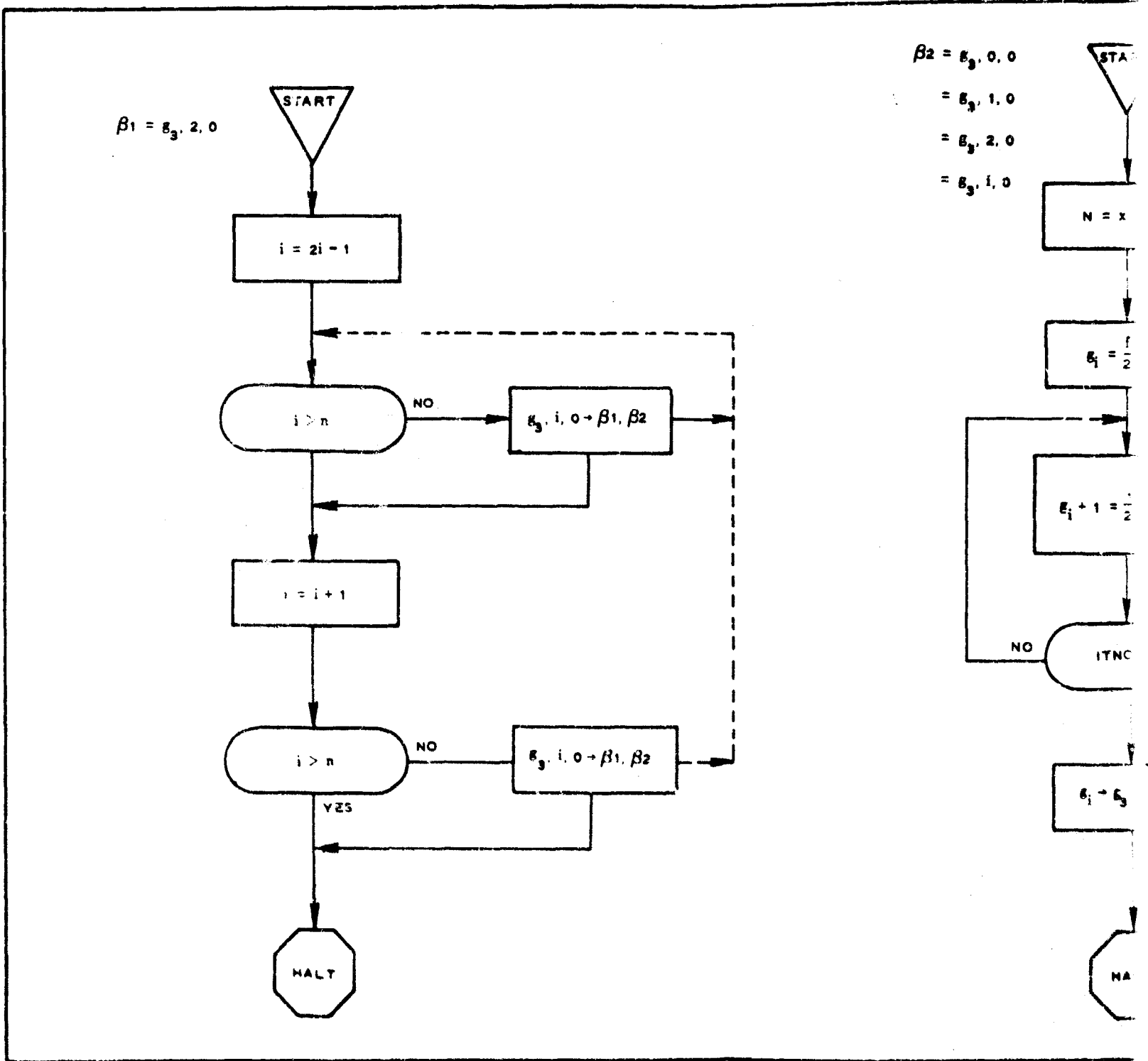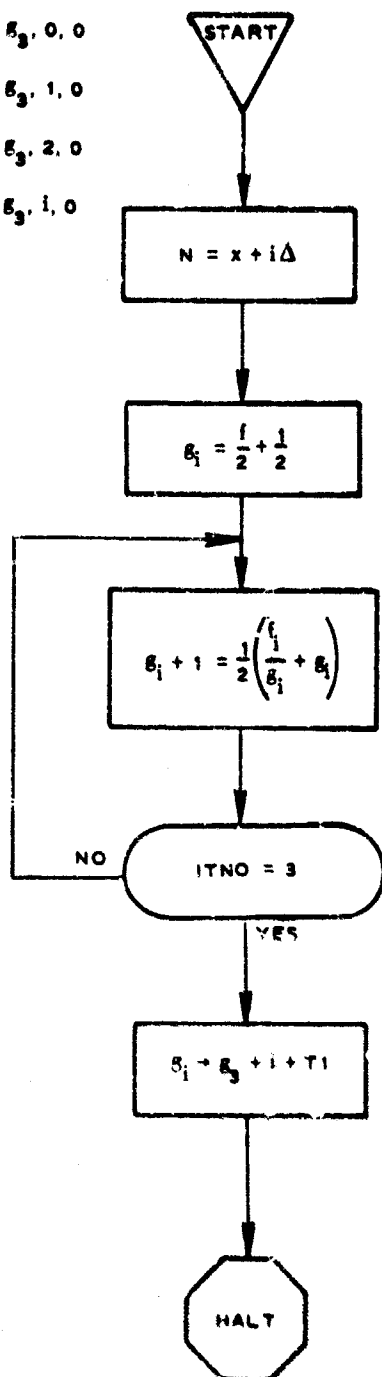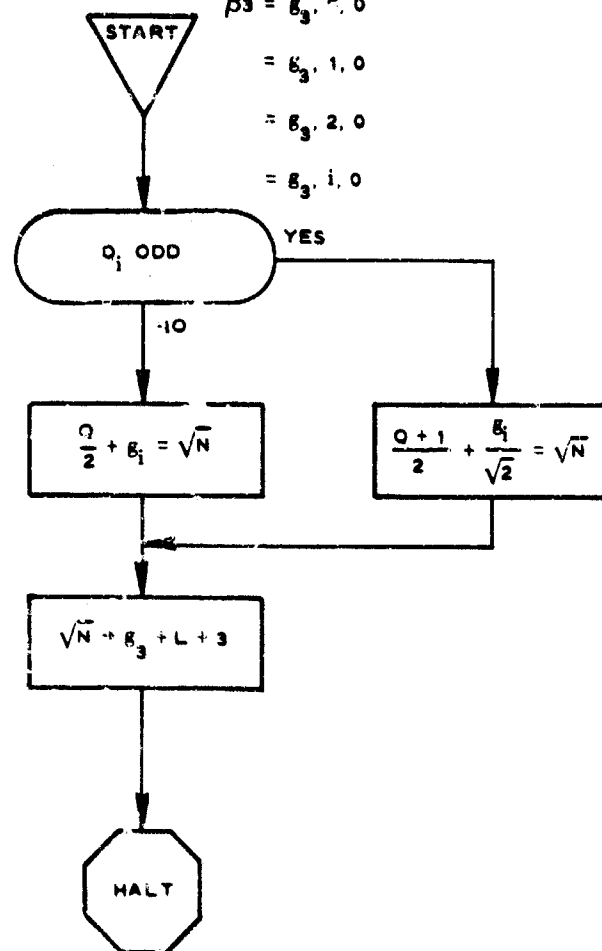
Figure III-3 - Activity Function 3 Flow Chart, Machine I

- 51 -

## TABLE III-4 - ACTIVITY FUNCTION 3 PROGRAM,

## MACHINE I

| Item | Instruction | | Remarks | Time ($\mu$sec) |
|------|-------------|------|---------|------------|
| | LDI | $\beta 1$ | | 30 |
| | JMP | INIT | | |
| | | | $\beta 1$ | |
| INIT | LDA | L1 | B1, B2, B3 | 30 |
| | BGN | $\beta 1$ | $g_3$, 2, 0 | |
| | ENB2 | $\emptyset$ | | 30 |
| | ENB4 | 2 | | |
| INIT 1 | STI | $\beta 2$ | | 30 |
| | LDA | L$\beta 2$ | | |
| | BGN | $\beta 2$ | | 30 |
| | INB2 | 1 | | |
| | BJP4 | INIT 1 | | 30 |
| | HLT | | | |
| | LDI | $\beta 1$ | | 30 |
| | JMP | I | | |
| | | | $\beta 1$ | |
| | | | B1, B2, B3 | |
| | | | $g_3$, 2, 0 | |
| I | SEH | $\beta 1$ | | 30 |
| | LDI | $\beta 1$ | | |
| | INB2 | 2 -1 | $i = 2i - 1$ | 30 |
| | ENA | 2 $\emptyset$ | | |
| | COM  1 | 2 | $2i - 1:n$ | 30 |
| | HLT | | $\geq$ | |
| | STI | $\beta 1$ | $\leq$ | 30 |
| | LDA | L$\beta 1$ | | |
| | BGN | $\beta 1$ | $\beta 1$ | 30 |
| | STI | $\beta 2$ | B1, B2, B3 | |

## TABLE III-4 - ACTIVITY FUNCTION 3 PROGRAM,

### MACHINE I (Continued)

| Item | Instruction | | Remarks | Time (μsec) |
|------|------|------|---------|-------------|
| | LDA | L$\beta$2 | $g_3$, 2i - 1, 0 | 30 |
| | BGN | $\beta$2 | | |
| | ENA | 1 | i = 2i | |
| | COM 1 | 2 | 2i:n | |
| | HLT | | $\geq$ | 30 |
| | STI | $\beta$1 | $\leq$ | |
| | LDA | L$\beta$1 | $\beta$1 | 30 |
| | BGN | $\beta$1 | B1, B2, B3 | |
| | STI | $\beta$2 | $g_3$, 2i, 0 | 30 |
| | LDA | L$\beta$2 | | |
| | BGN | $\beta$2 | | 30 |
| | HLT | | | |
| L$\beta$1 | LDI | . . . | | |
| | JMP | I | | |
| L$\beta$2 | LDI | . . . | | |
| | JMP | LOOP | | |
| | LDI | $\beta$2 | | 30 |
| | JMP | LOOP | | |
| | | | $\beta$2 | |
| | | | B1, B2, B3 | |
| | | | $g_3$, 0, 0 | |
| LOOP | SEH | $\beta$2 | $g_3$, 1, 0 | 30 |
| | LDI | $\beta$2 | $g_3$, 2, 0 | |
| | STI | $\beta$2 | $g_3$, i, 0 | 30 |
| | LDA | L$\beta$3 | | |

## TABLE III-4 - ACTIVITY FUNCTION 3 PROGRAM, MACHINE I (Continued)

| Item | Instruction | | | | Remarks | Time ($\mu$sec) |
|------|------|---|---|------|---------|-----------|
| | BGN | | | $\beta3$ | | 30 |
| | ENA | 2 | | $\phi$ | | |
| | FMP | 1 | | 1 | i$\Delta$ | 60 |
| | FAD | 1 | | $\phi$ | | |
| | STA | 1 | 2 | T2 | ni | 30 |
| | AND | | | f MASK | | |
| | STA | 1 | 2 | T3 | f | 30 |
| | ARS | | | 1 | | |
| | ADD | | | 1/2 | | 30 |
| | STA | 1 | 2 | T1 | | |
| | ENB4 | | | 2 | | 30 |
| | NOD | | | | | |
| LOOP 1 | LDA | 1 | 2 | T3 | | 60 |
| | FDV | 1 | 2 | T1 | g | |
| | SEH | 1 | 2 | T1 | | 60 |
| | FAD | 1 | 2 | T1 | | |
| | FDV | | | 2 | | 60 |
| | STA | 1 | 2 | T1 | | |
| | BJP4 | | | LOOP 1 | | 30 |
| | HLT | | | | | |
| L3 | LDI | | | . . . | | 30 |
| | JMP | | | Q | | |
| | LDI | | | $\beta3$ | | 30 |
| | JMP | | | Q | | |
| Q | SEH | | | $\beta3$ | | 30 |
| | LDI | | | $\beta3$ | | |
| | LDA | 1 | 2 | T2 | | 30 |
| | AND | | | QIMSK | Exponent Bit 1 | |

TABLE III-4 - ACTIVITY FUNCTION 3 PROGRAM,

MACHINE I (Continued)

| Item | Instruction | | | Remarks | Time (μsec) |
|------|------|------|------|------|------|
| | JNZA | | QA | $Q1 \neq 0$ | 30 |
| | LDA | 1 2 | T2 | $Q1 \neq 0$ | |
| | AND | | QMSK | Exponent | 30 |
| | NOP | | | | |
| QB | ARS | | 1 | | 30 |
| | ADD | 1 2 | T1 | Fraction | |
| | STA | 1 2 | 2 | $\sqrt{N}$ | 30 |
| | HLT | | | | |
| | LDA | 1 2 | T2 | | 30 |
| | AND | | QMSK | | |
| | ADD | | Q1 | | 30 |
| | JMP | | QB | | |

temporary vectors, T1, T2, and T3, are generated to hold (1) the initial calculated guess of the square root, (2) the number itself, and (3) the fractional part of the number.

Temporary storage appears to be sizeable. The temporary addresses are actually reserved addresses not necessarily occupied. Only a portion of this block would be filled at any one time, since processors operating on the program are continually started and stopped as data are entered, used, and erased. The addresses when occupied are not available for use by other processors. However, when the data is erased the location is then free.

In a conventional memory, temporary storage is defined as a certain block of words occupied at a certain time. This area cannot be used otherwise to store instructions or data.

## TABLE III-5 - ACTIVITY FUNCTION 3 DATA VECTOR

## FORMAT, MACHINE 1

| Function $g_3(x)$ | | Temporary vectors | | |
|---|---|---|---|---|
| Address | Data | T1 | T2 | T3 |
| $g_3$ | x | | | |
| $g_3 + 1$ | $\Delta$ | | | |
| $g_3 + 2$ | n | | | |
| $g_3 + 3$ | $\sqrt{x_0}$ | $g_0$ | $N_0$ | $f_0$ |
| $g_3 + 4$ | $\sqrt{x_1}$ | $g_1$ | $N_1$ | $f_1$ |
| $g_3 + 5$ | $\sqrt{x_2}$ | $g_2$ | $N_2$ | $f_2$ |
| $g_3 + 6$ | $\sqrt{x_3}$ | $g_3$ | $N_3$ | $f_3$ |
| $g_3 + 7$ | $\sqrt{x_4}$ | $g_4$ | $N_4$ | $f_4$ |
| $g_3 + 8$ | $\sqrt{x_5}$ | $g_5$ | $N_5$ | $f_5$ |
| $g_3 + 9$ | $\sqrt{x_6}$ | $g_6$ | $N_6$ | $f_6$ |
| $g_3 + 10$ | $\sqrt{x_7}$ | $g_7$ | $N_7$ | $f_7$ |
| $g_3 + 11$ | $\sqrt{x_8}$ | $g_8$ | $N_8$ | $f_8$ |
| $g_3 + 12$ | $\sqrt{x_9}$ | $g_9$ | $N_9$ | $f_9$ |
| $g_3 + 13$ | $\sqrt{x_{10}}$ | $g_{10}$ | $N_{10}$ | $f_{10}$ |
| $g_3 + 14$ | $\sqrt{x_{11}}$ | $g_{11}$ | $N_{11}$ | $f_{11}$ |
| $g_3 + 15$ | $\sqrt{x_{12}}$ | $g_{12}$ | $N_{12}$ | $f_{12}$ |
| $g_3 + 16$ | $\sqrt{x_{13}}$ | $g_{13}$ | $N_{13}$ | $f_{13}$ |
| $g_3 + 17$ | $\sqrt{x_{14}}$ | $g_{14}$ | $N_{14}$ | $f_{14}$ |
| $g_3 + 18$ | $\sqrt{x_{15}}$ | $g_{15}$ | $N_{15}$ | $f_{15}$ |
| $g_3 + 19$ | $\sqrt{x_{16}}$ | $g_{16}$ | $N_{16}$ | $f_{16}$ |
| $g_3 + 20$ | $\sqrt{x_{17}}$ | $g_{17}$ | $N_{17}$ | $f_{17}$ |
| $g_3 + 21$ | $\sqrt{x_{18}}$ | $g_{18}$ | $N_{18}$ | $f_{18}$ |
| $g_3 + 22$ | $\sqrt{x_{19}}$ | $g_{19}$ | $N_{19}$ | $f_{19}$ |
| $g_3 + 23$ | $\sqrt{x_{20}}$ | $g_{20}$ | $N_{20}$ | $f_{20}$ |

In the Machine I memory a portion of each word is used to denote the name or address of the word. Any word can have any name that is representable in the 24 bits of the name field. In addition, any unnamed word is available for use by any program at any time.

It is possible through indiscriminate naming to have common names in a number of programs, which situation may be undesirable. Hence, where a large number of programs are running, a portion of the name field, the prefix, should be used to isolate names to a particular program. Such a prefix name is unique to a particular program. The original name plus the prefix constitute a unique name for the individual program.

During the study, the unique prefix names were carried in index register 1. The other indices and address fields of the instructions were used to generate the suffix names.

### d. Activity Function 4

The flow chart for the $g_4(x) = 2 \sin x$ activity function is shown in Figure III-4; the program, in Table III-6; and the data vector format, in Table III-7.

This function is composed of six subprogram-- HSIN, SER, SIN $(i + j)\Delta$, COS $(i + j)\Delta$, SIN $(x + j\Delta)$ and COS $(x + j\Delta)$. HSIN computes $a$ and $a^*$ for the input, $x$, and also $\delta$ and $\delta^*$ for the input, $\Delta$; $a$, $a^*$, $\delta$, and $\delta^*$ are inputs to SER for the calculation of $\sin x$, $\cos x$, $\sin \Delta$, and $\cos \Delta$.

Sin $2\Delta$ and cos $2\Delta$ are calculated, and a tree is started to generate indices that permit calculation of $\sin i\Delta$, $\cos i\Delta$, $\sin (i - 1)\Delta$, and $\cos (i - 1)\Delta$ for $i > 2$. For each incremental angle the $\sin x + i\Delta$ and $\cos x + i\Delta$ are computed and stored in the output data vector.

To start the SIN routine, index $g_4$, 0, 0 and index $g_4 + 1$, 5, 0 are transferred to two available processors. These units execute the HSIN subprogram, one working with $x$ and the other working with $\Delta$. After $a$, $a^*$, $\delta$, and $\delta^*$ are calculated, four processors are started, each executing the SER subpr. The outputs are $\sin x$, $\cos x$, $\sin \Delta$ and $\cos \Delta$. The
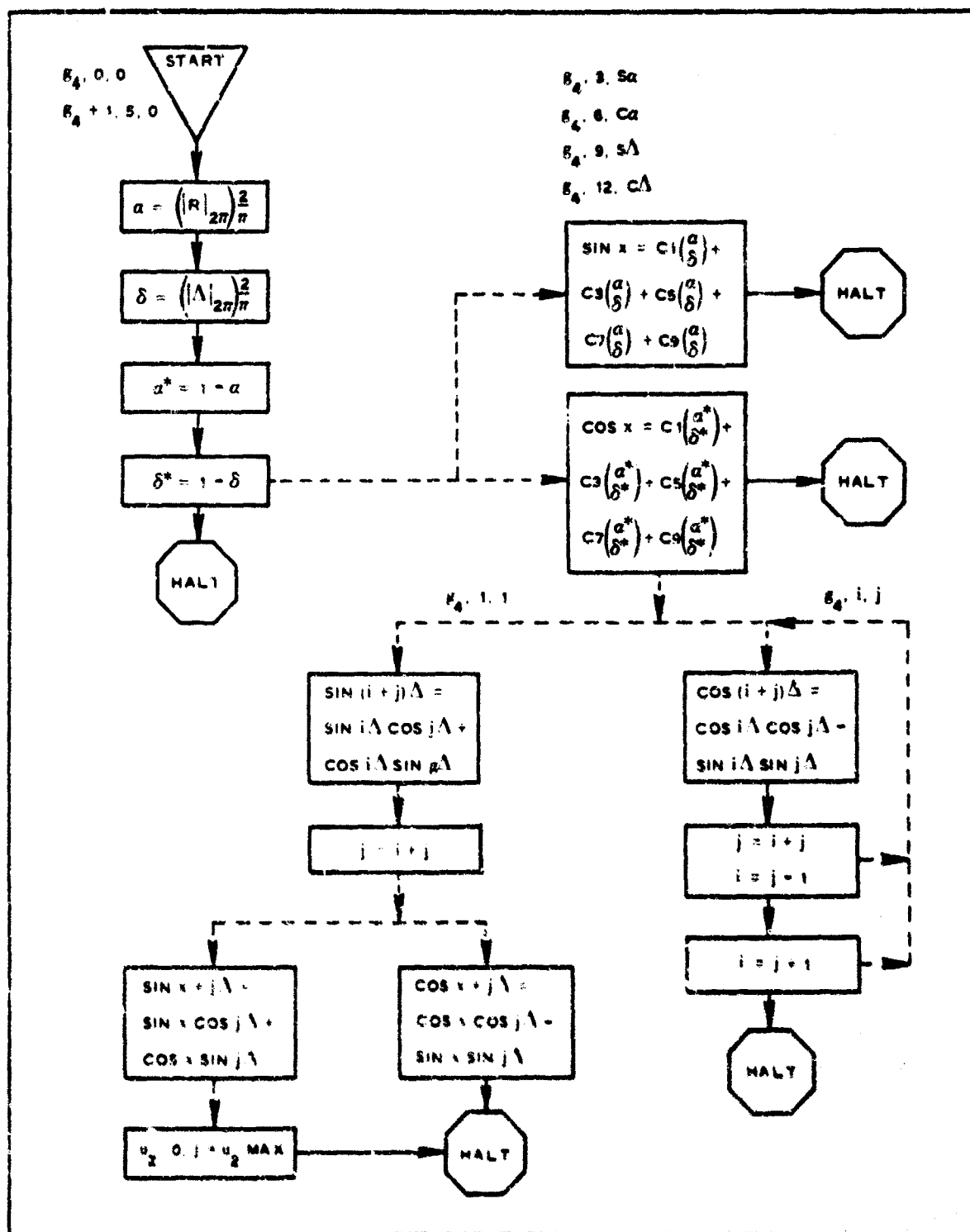
Figure III-4 - Activity Function 4 Flow Chart, Machine I

TABLE III-6.--ACTIVITY FUNCTION 4 PROGRAM, MACHINE I

| Item | Instruction | | | | Remarks | Time ($\mu$sec) |
|------|-----|---|---|---|---------|-----------|
|      | LDI |   |   | $\rho$ | $\rho$ | 30 |
|      | JMP |   |   | HSIN | B1, B2, B3 | |
|      |     |   |   |      | $8_4$, 0, 0 | |
|      |     |   |   |      | $8_{4+1}$, 5, 0 | |
| HSIN | SEH |   |   | $\rho$ | | 30 |
|      | LDI |   |   | $\rho$ | | |
|      | LDA | 1 |   | $\phi$ | x or $\Delta$ | 30 |
|      | NOP |   |   |      | | |
| HS   | FSU |   |   | $2\pi$ | | 60 |
|      | NOP |   |   |      | | |
|      | COM |   |   | $2\pi$ | | 30 |
|      | JMP |   |   | HS   | | |
|      | STA | 1 | 2 | 3    | R or r | 30 |
|      | NHA |   |   | $\gamma$ | | |
|      | ARS |   |   | 1    | $\gamma$ A | 30 |
|      | STQ | 1 | 2 | 5    | $\pm$ sin x, $\pm$ sin $\Delta$ | |
|      | LRS |   |   | 1    | | 30 |
|      | STQ | 1 | 2 | 8    | cos x, $\pm$ cos $\Delta$ | |
| HS2  | SEH | 1 | 2 | 3    | | 30 |
|      | LDA | 1 | 2 | 3    | R or r | |
|      | FMP |   |   | $2/\pi$ | | 60 |
|      | STA | 1 | 2 | 3    | $a$ or $\delta$ | |
|      | LAC | 1 | 2 | 3    | | 30 |
|      | STA | 1 | 2 | 6    | $a^*$ or $\delta^*$ | |
|      | BJP3 |  |   | HLT  | | 30 |
|      | NOP |   |   |      | | |
| HS1  | ENB4 |  |   | 3    | | 30 |
|      | NOP |   |   |      | $\beta$ output | |

## TABLE III-6 - ACTIVITY FUNCTION 4 PROGRAM, MACHINE I (Continued)

| Item | Instruction | | | | Remarks | Time ($\mu$sec) |
|------|------|---|---|---|---------|-----------|
| HS3 | LDI | 4 | | k | B1, B2, B3 | 30 |
| | STI | | | $\beta$ | k, $g_4$, 3, S$\alpha$ | |
| | LDA | | | L | +1, $g_4$, 6, C$\alpha$ | 30 |
| | BGN | | | $\beta$ | +2, $g_4$, 9, S$\Delta$ | |
| | BJP4 | | | HS3 | +3, $g_4$, 12, C$\Delta$ | 30 |
| | HLT | | | | | |
| L | LDI | | | . . . | | |
| | JMP | | | SER | | |
| $\gamma$ | $2\pi$ | | | | | |
| $\gamma$ | $3\pi/2$ | | | + - | | |
| $\gamma$ | $\pi$ | | | - + | | |
| $\gamma$ | $\pi/2$ | | | + + | | |
| $2\pi$ | | | | | | |
| $\pi/2$ | | | | | | |
| | LDI | | | $\beta$ | | 30 |
| | JMP | | | SER | $\beta$ | |
| | | | | | B1, B2, B3 | |
| | | | | | $g_4$, 3, S | |
| SER | SEH | | | $\beta$ | $g_4$, 6, C | 30 |
| | LDI | | | $\beta$ | $g_4$, 9, S | |
| | LDA | 1 | 2 | $\phi$ | $g_4$, 12, C | 60 |
| | FMP | 1 | 2 | $\phi$ | | |
| | STA | 1 | 2 | 1 | | 60 |
| | FMP | | | C9 | | |
| | FAD | | | C7 | | 60 |
| | FMP | 1 | 2 | 1 | | |
| | FAD | | | C5 | | 60 |
| | FMP | 1 | 2 | 1 | | |

## TABLE III-6 - ACTIVITY FUNCTION 4 PROGRAM, MACHINE I (Continued)

| Item | Instruction | | | | Remarks | Time (μsec) |
|------|------|---|---|------|---------|-------------|
| | FAD | | | C3 | | 60 |
| | FMP | 1 | 2 | 1 | | |
| | FAD | | | C1 | | 60 |
| | FMP | 1 | 2 | ∅ | | |
| | STA | 1 | 3 | ∅ | | 30 |
| | LDA | 1 | 2 | 2 | | |
| | AND | | | SMSK | | 30 |
| | SEH | 1 | 3 | ∅ | | |
| | OR | 1 | 3 | ∅ | | 30 |
| | STA | 1 | 3 | ∅ | ± sin x, cos x, sin Δ, cos Δ | |
| | INB2 | | | -12 | | 30 |
| | BJP2 | | | SER1 | | |
| | HLT | | | | | 30 |
| | NOP | | | | | |
| SER1 | ENB2 | | | 1 | START SIN (i + j)Δ | 30 |
| | ENB3 | | | 1 | COS (i + j)Δ | |
| | STI | | | m | | 30 |
| | STI | | | n | | |
| | LDA | | | N | | 30 |
| | BGN | | | m | | |
| SER2 | LDA | | | Sα | | 30 |
| | NP2 | | | SER2 | | |
| | SEH | | | Sα | | 30 |
| | FAD | | | Sα | | |
| | STA | | | Sα | | 30 |
| | NOP | | | | | |
| SER3 | LDA | | | Cα | | 30 |
| | NPJ | | | SER3 | | |

TABLE III-6 — ACTIVITY FUNCTION 4 PROGRAM, MACHINE I (Continued)

| Item | Instruction | | | | Remarks | Time ($\mu$sec) |
|------|------|---|---|---|---------|-----------|
| | SEH | | | $C\alpha$ | | 30 |
| | FAD | | | $C\alpha$ | | |
| | STA | | | $C\alpha$ | | 30 |
| | LDA | | | $N + 1$ | | |
| | BGN | | | n | | 30 |
| | ENB2 | | | $\emptyset$ | | |
| | STI | | | g | | 30 |
| | LDA | | | P | START | |
| | BGN | | | g | SIN $(x + j\Delta)$, $j = 1$ | 30 |
| | STI | | | h | COS $(x + j\Delta)$, $j = 1$ | |
| | LDA | | | $P + 1$ | | 30 |
| | BGN | | | h | | |
| | HLT | | | | | 30 |
| | NOP | | | | | |
| N | LDI | | | . . . | | |
| | JMP | | | SIN $(i + j)\Delta$ | | |
| N + 1 | LDI | | | . . . | | |
| | JMP | | | COS $(i + j)\Delta$ | | |
| | LDI | | | m | | 30 |
| | JMP | | | SIN $(i + j)\Delta$ | | |
| SIN $(i + j)\Delta$ | SEH | | | m | | 30 |
| | ADD | | | m | B1, B2, B3 | |
| SijA | LDA | 1 | 2 | $S\Delta$ | $g_4$, i, j | 30 |
| | NPJ | | | SijA | | |
| SijB | FMP | 1 | 3 | $C\Delta$ | | 60 |
| | NPJ | | | SijB | | |
| | STA | 1 | 2 | 3 $S\Delta T$ | $(i + j)$TEMP | 30 |
| | NOP | | | | | |
| SijC | LDA | 1 | | 3 $S\Delta$ | | 30 |

## TABLE III-6 - ACTIVITY FUNCTION 4 PROGRAM, MACHINE I (Continued)

| Item | Instruction | | | | | Remarks | Time ($\mu$sec) |
|------|-----|---|---|---|------|---------|-----------|
| | NPJ | | | | SijC | | |
| SijD | FMP | 1 | 2 | | C$\Delta$ | | 60 |
| | NPJ | | | | SijD | | |
| | SEH | 1 | 2 | 3 | S$\Delta$T | | 30 |
| | ADD | 1 | 2 | 3 | $\bar{S}\Delta$T | | |
| | STA | 1 | 2 | 3 | S$\Delta$ | | 30 |
| | INB3 | | 2 | | $\phi$ | | |
| | STI | | | | g | g or table | 30 |
| | LDA | | | | P | | |
| | BGN | | | | g | B1, B2, B3 | 30 |
| | STI | | | | L | $g_4$, 0, j | |
| | LDA | | | | P + 1 | | 30 |
| | BGN | | | | h | | |
| | HLT | | | | | | 30 |
| | NOP | | | | | | |
| P | LDI | | | | . . . | g | |
| | JMP | | | | SIN $(x + j\Delta)$ | | |
| P + 1 | LDI | | | | . . . | h | |
| | JMP | | | | COS $(x + j\Delta)$ | | |
| | LDI | | | | n | | 30 |
| | JMP | | | | COS $(i + j)\Delta$ | | |
| | | | | | | B1, B2, B3 | |
| | | | | | | $g_4$, i, j | |
| COS $(i + j)\Delta$ | SEH | | | | n | | 30 |
| | ADD | | | | n | | |
| | LDA | 1 | 2 | | S$\Delta$ | SIN i$\Delta$ | 30 |
| | NPJ | | | | | | |
| | FMP | 1 | | 3 | S$\Delta$ | SIN j$\Delta$ | 60 |
| | NPJ | | | | | | |

## TABLE III-6 - ACTIVITY FUNCTION 4 PROGRAM, MACHINE I (Continued)

| Item | Instruction | | | | | Remarks | Time (µsec) |
|------|-----|---|---|---|------|---------|------------|
| | STA | 1 | 2 | 3 | CΔT | (i + j)TEMP | 30 |
| | LDA | 1 | 2 | | CΔ | | |
| | NPJ | | | | | | 60 |
| | FMP | 1 | | 3 | CΔ | | |
| | NPJ | | | | | | 30 |
| | SEH | 1 | 2 | 3 | CΔT | | |
| | SUB | 1 | 2 | 3 | CΔT | | 30 |
| | STA | 1 | 2 | 3 | CΔ | | |
| | INB3 | | 2 | | ∅ | i + j | 30 |
| | ENB2 | | | 3 | -1 | i + j - 1 | |
| | LAC | 1 | | | 2 | -n | 30 |
| | INA | | 2 | 3 | ∅ | | |
| | JNGA | | | | CA | | 30 |
| | JNZA | | | | HLT | | |
| CA | STI | | | | e | | 30 |
| | LDA | | | | M | | |
| | BGN | | | | e | LDI___, JMP, SijA | 30 |
| | STI | | | | f | | |
| | LDA | | | | m + 1 | | 30 |
| | BGN | | | | f | LDI___, JMP, CijΔ | |
| | INB2 | | | | 1 | | 30 |
| | LAC | 1 | | | 2 | -n | |
| | INA | | 2 | 3 | ∅ | | 30 |
| | JNGA | | | | CB | | |
| | JNZA | | | | | | 30 |
| | NOP | | | | | | |
| CB | STI | | | | e | | 30 |
| | LDA | | | | M | LDI___, JMP SijΔ | |
| | BGN | | | | e | | 30 |

## TABLE III-6 - ACTIVITY FUNCTION 4 PROGRAM, MACHINE I (Continued)

| Item | Instruction | | | Remarks | Time (μsec) |
|---|---|---|---|---|---|
| | STI | | f | | |
| | LDA | | m + 1 | | 30 |
| | BGN | | f | LDI___, JMP Cij$\Delta$ | |
| | HLT | | | | 30 |
| | NOP | | | | |
| | LDi | | g | | |
| | JMP | | SIN x + j$\Delta$ | | |
| SIN (x + j$\Delta$) | SEH | | g | B1, B2, B3 | 30 |
| | ADD | | g | $g_4$, 0, j | |
| | LDA | | S$\alpha$ | | 30 |
| | FMP | 1 | 3 C$\Delta$ | COS j$\Delta$ | |
| | NPJ | | | | 30 |
| | STA | 1 | 3 S$\alpha$T | | |
| | LDA | | C$\alpha$ | | 60 |
| | FMP | 1 | 3 S$\Delta$ | | |
| | SEH | 1 | 3 S$\alpha$T | | 30 |
| | ADD | 1 | 3 S$\alpha$T | | |
| | STA | 1 | 3 S$\alpha$ | SIN x + j$\Delta$ | 30 |
| | LDA | | GAM | | |
| | INAL | 3 | 3 | | 30 |
| | STA | 3 | mtem1 | | |
| | LDA | | GIM + 1 | START $u_2$ MAX | 30 |
| | BGN | ? | mtem1 | | |
| | HLT | | | | 30 |
| | NOP | | | | |
| G4m | $u_2$ | | 0 | | |
| | 0 | | | | |
| | LDI | | h | | 30 |

TABLE III-6 - ACTIVITY FUNCTION 4 PROGRAM, MACHINE I (Continued)

| Item | Instruction | | | Remarks | Time ($\mu$sec) |
|---|---|---|---|---|---|
| | JMP | | COS $(x + j\Delta)$ | | 30 |
| COS $(x + j\Delta)$ | SEH | | h | | 30 |
| | ADD | | h | | |
| | LDA | | $S\alpha$ | | 60 |
| | FMP | 1 | 3 $S\Delta$ | sin $j\Delta$ | |
| | STA | 1 | 3 $S\alpha T$ | | 30 |
| | LD | | $C\alpha$ | | |
| | FMP | 1 | 3 $C\Delta$ | cos $j\Delta$ | 60 |
| | NPJ | | | | |
| | SEH | 1 | 3 $S\alpha T$ | | 30 |
| | SUB | 1 | 3 $S\alpha T$ | | |
| | STA | 1 | 3 $C\alpha$ | COS $(x + j\Delta)$ | 30 |
| | HLT | | | | |

indices transferred to the SER processors are $g_4$, 3, $S\alpha$; $g_4$, 6, $C\alpha$; $g_4$, 9, $S\Delta$; and $g_4$, 12, $C\Delta$; $g_4$ is the prefix name; 3, 6, 9, and 12 are data vector addresses, relative to $g_4$, of the input variables $\alpha$, $\alpha^*$, $\delta$, and $\delta^*$; $S\alpha$, $C\alpha$, $S\Delta$, and $C\Delta$ are initial addresses of the output data vectors.

The SER subprogram generates one set of indices to start the SIN $(i + j)\Delta$ and COS $(i + j)\Delta$ subprograms and also generates one index set to start the calculation of sin $x + \Delta$ and cos $x + \Delta$. The index set that starts the SIN $(i + j)\Delta$ and COS $(i + j)\Delta$ subprograms is used to generate the tree of indices that are, in turn, inputs to the SIN $(i + j)\Delta$, COS $(i + j)\Delta$, SIN $(x + j\Delta)$, and COS $(x + j\Delta)$ subprograms.

The indices transferred to the processor executing the HSIN subprogram are erased, and x or $\Delta$ is then normalized mod $2\pi$. The normalized quantity is next compared to the four quadrant angles, each of which has stored in its lower two bits the correct signs of sines and cosines of angles

## TABLE III-7 - ACTIVITY FUNCTION 4 DATA VECTOR FORMAT, MACHINE I

| Address | Data | Address | Data | Address | Data | Address | Data | Address | Data | T1 | T2 | T3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Function $g_4(x)$ | | | | | Temporary vectors | | |
| $S_4$ | $x$ | $S\alpha$ | $\sin \alpha$ | $C\alpha$ | $\cos \alpha$ | $S\Delta$ | $\cdots$ | $C\Delta$ | $\cdots$ | T | ST | CT |
| $S_4+1$ | $\Delta$ | $S\alpha+1$ | $\sin \alpha+1\Delta$ | $C\alpha+1\Delta$ | $\cos \alpha+\Delta$ | $S\Delta+1$ | $\sin \Delta$ | $C\Delta+1$ | $\cos \Delta$ | T | ST | CT |
| $S_4+2$ | $n$ | $S\alpha+2$ | $\sin \alpha+2\Delta$ | $C\alpha+2\Delta$ | $\cos \alpha+2\Delta$ | $S\Delta+2$ | $\sin 2\Delta$ | $C\Delta+2$ | $\cos 2\Delta$ | T | ST | CT |
| $S_4+3$ | $a^2$ | $S\alpha+3$ | $\sin \alpha+3\Delta$ | $C\alpha+3\Delta$ | $\cos \alpha+3\Delta$ | $S\Delta+3$ | $\sin 3\Delta$ | $C\Delta+3$ | $\cos 3\Delta$ | T | ST | CT |
| $S_4+4$ | SIGN | $S\alpha+4$ | $\sin \alpha+4\Delta$ | $C\alpha+4\Delta$ | $\cos \alpha+4\Delta$ | $S\Delta+4$ | $\sin 4\Delta$ | $C\Delta+4$ | $\cos 4\Delta$ | T | ST | CT |
| $S_4+5$ | $a^*$ | $S\alpha+5$ | $\sin \alpha+5\Delta$ | $C\alpha+5\Delta$ | $\cos \alpha+5\Delta$ | $S\Delta+5$ | $\sin 5\Delta$ | $C\Delta+5$ | $\cos 5\Delta$ | T | ST | CT |
| $S_4+6$ | $a^{*2}$ | $S\alpha+6$ | $\sin \alpha+6\Delta$ | $C\alpha+6\Delta$ | $\cos \alpha+6\Delta$ | $S\Delta+6$ | $\sin 6\Delta$ | $C\Delta+6$ | $\cos 6\Delta$ | T | ST | CT |
| $S_4+7$ | SIGN | $S\alpha+7$ | $\sin \alpha+7\Delta$ | $C\alpha+7\Delta$ | $\cos \alpha+7\Delta$ | $S\Delta+7$ | $\sin 7\Delta$ | $C\Delta+7$ | $\cos 7\Delta$ | T | ST | CT |
| $S_4+8$ | $\delta$ | $S\alpha+8$ | $\sin \alpha+8\Delta$ | $C\alpha+8\Delta$ | $\cos \alpha+8\Delta$ | $S\Delta+8$ | $\sin 8\Delta$ | $C\Delta+8$ | $\cos 8\Delta$ | T | ST | CT |
| $S_4+9$ | $\delta^2$ | $S\alpha+9$ | $\sin \alpha+9\Delta$ | $C\alpha+9\Delta$ | $\cos \alpha+9\Delta$ | $S\Delta+9$ | $\sin 9\Delta$ | $C\Delta+9$ | $\cos 9\Delta$ | T | ST | CT |
| $S_4+10$ | SIGN | $S\alpha+10$ | $\sin \alpha+10\Delta$ | $C\alpha+10\Delta$ | $\cos \alpha+10\Delta$ | $S\Delta+10$ | $\sin 10\Delta$ | $C\Delta+10$ | $\cos 10\Delta$ | T | ST | CT |
| $S_4+11$ | $\delta^*$ | $S\alpha+11$ | $\sin \alpha+11\Delta$ | $C\alpha+11\Delta$ | $\cos \alpha+11\Delta$ | $S\Delta+11$ | $\sin 11\Delta$ | $C\Delta+11$ | $\cos 11\Delta$ | T | ST | CT |
| $S_4+12$ | $\delta^{*2}$ | $S\alpha+12$ | $\sin \alpha+12\Delta$ | $C\alpha+12\Delta$ | $\cos \alpha+12\Delta$ | $S\Delta+12$ | $\sin 12\Delta$ | $C\Delta+12$ | $\cos 12\Delta$ | T | ST | CT |
| $S_4+13$ | SIGN | $S\alpha+13$ | $\sin \alpha+13\Delta$ | $C\alpha+13\Delta$ | $\cos \alpha+13\Delta$ | $S\Delta+13$ | $\sin 13\Delta$ | $C\Delta+13$ | $\cos 13\Delta$ | T | ST | CT |
| $S_4+14$ | | $S\alpha+14$ | $\sin \alpha+14\Delta$ | $C\alpha+14\Delta$ | $\cos \alpha+14\Delta$ | $S\Delta+14$ | $\sin 14\Delta$ | $C\Delta+14$ | $\cos 14\Delta$ | T | ST | CT |
| $S_4+15$ | | $S\alpha+15$ | $\sin \alpha+15\Delta$ | $C\alpha+15\Delta$ | $\cos \alpha+15\Delta$ | $S\Delta+15$ | $\sin 15\Delta$ | $C\Delta+15$ | $\cos 15\Delta$ | T | ST | CT |
| $S_4+16$ | | $S\alpha+16$ | $\sin \alpha+16\Delta$ | $C\alpha+16\Delta$ | $\cos \alpha+16\Delta$ | $S\Delta+16$ | $\sin 16\Delta$ | $C\Delta+16$ | $\cos 16\Delta$ | T | ST | CT |
| $S_4+17$ | | $S\alpha+17$ | $\sin \alpha+17\Delta$ | $C\alpha+17\Delta$ | $\cos \alpha+17\Delta$ | $S\Delta+17$ | $\sin 17\Delta$ | $C\Delta+17$ | $\cos 17\Delta$ | T | ST | CT |
| $S_4+18$ | | $S\alpha+18$ | $\sin \alpha+18\Delta$ | $C\alpha+18\Delta$ | $\cos \alpha+18\Delta$ | $S\Delta+18$ | $\sin 18\Delta$ | $C\Delta+18$ | $\cos 18\Delta$ | T | ST | CT |
| $S_4+19$ | | $S\alpha+19$ | $\sin \alpha+19\Delta$ | $C\alpha+19\Delta$ | $\cos \alpha+19\Delta$ | $S\Delta+19$ | $\sin 19\Delta$ | $C\Delta+19$ | $\cos 19\Delta$ | T | ST | CT |
| $S_4+20$ | | $S\alpha+20$ | $\sin \alpha+20\Delta$ | $C\alpha+20\Delta$ | $\cos \alpha+20\Delta$ | $S\Delta+20$ | $\sin 20\Delta$ | $C\Delta+20$ | $\cos 20\Delta$ | T | ST | CT |

within that quadrant. The NHA $\gamma$ instruction obtains the smallest quadrant angle larger than the argument in the accumulator and replaces the contents of the accumulator. The sign bits are shifted into Q and stored.

Now $a^*$ and $\delta^*$ are computed, and the indices to be transferred to the SER program are stored prior to starting the four SER processing units. The SER program computes sin x, cos x, sin $\Delta$, and cos $\Delta$ from inputs $\alpha$, $a^*$, $\delta$, and $\delta^*$ using the Hastings Sine series. The processing unit that computes cos $\Delta$ generates indices and starts instructions for the processors to begin executing the SIN $(i + j)\Delta$, COS $(i + j)\Delta$, SIN $(x + j\Delta)$, and COS $(x + j\Delta)$ subprograms.

The SIN $(i + j)\Delta$ subprogram gets its input indices from a temporary table, after which the indices are erased. The product sin $i\Delta$, cos $j\Delta$ is found and added to sin $j\Delta$ cos $i\Delta$ and stored. Nonpresence jump instructions are used to be certain that the words fetched from memory are the exact ones requested. The SIN $(i + j)\Delta$ subprogram also generates indices, which are transferred to processors assigned to the SIN $(x + j\Delta)$ and COS $(x + j\Delta)$ subprograms.

The COS $(i + j)\Delta$ subprogram gets its index inputs from a temporary table in a similar manner to the SIN $(i + j\Delta)$ subprogram. While the computation is similar to that for the SIN $(i + j\Delta)$ subprogram, the COS $(i + j\Delta)$ program generates indices that are used by the SIN $(i + j)\Delta$ and COS $(i + j)\Delta$ subprograms to generate mor $ \ni$ branches of the tree $x + j\Delta$. For each i, j input, 2 sets of indices are generated: $j = i + j$, $i = j - 1$ and $j = i + j$, $i = j$. Hence for $i = 1$, the inputs are $j = 1$ and the output sets are $i = 1$, $j = 2$ and $i = 2$, $j = 2$. These in turn generate $i = 2$, $j = 3$, $i = 3$, $j = 3$ and $i = 3$ $j = 4$, $i = 4$, $j = 4$.

For each set of indices sent to the SIN $(i + j)\Delta$ subprogram, the sum $j = i + j$ is sent to the SIN $(x + j\Delta)$ and COS $(x + j\Delta)$ subprograms. For each additional SIN $(x + j\Delta)$ subprogram generated, there is a processing unit assigned to the $u_2$ maximization program.

### e. Activity Function 5

The flow chart for the $g_5(x)$ activity function is shown in Figure III-5; the program, in Table III-8; and the data vector format, in Table III-9. The returns calculated from this function are

$$g_5(x) = \begin{pmatrix} 2x \text{ if } 0 \leq x \leq 1 \\ \\ 4 - 2x \text{ if } 1 \leq x \leq 2 \end{pmatrix}.$$

The $g_5(x)$ function is executed by transferring the address of the $g_5$ data vector to a processing unit and jumping to G5. Index registers 2 and 3 are initialized with 0 and n; x is obtained from the data vector and stored in (1) a temporary word at the end of the data vector and (2) the first word of the output portion of the data vector.

The iterative portion of the program begins by fetching and erasing the temporary word = x, adding $\Delta$ to it, and storing x + $\Delta$ back in the temporary location. The incremented value in A is now doubled and tested. If $0 \leq X \leq 1$, then the doubled value is stored in the data vector at the address B1 + B2 + 4. If $1 \leq X \leq 2$, then 4 - 2x is stored.

Index 2 is now compared to index 3. Index 2 carries the current iteration number, while index 3 carries the maximum number of iterations to be performed. If the maximum has not been exceeded, the program is repeated with index 2 incremented. If the maximum has been reached, the processing unit halts.

### f. Activity Function 6

The flow chart for activity function $g_6(x) = 2(x)$ is shown in Figure III-6; the program, in Table III-10; and the data vector format, in Table III-9, along with the $g_5(x)$ format.

This function is started by transferring the address of the $g_6$ data vector to a processing unit and jumping to G6; x is obtained from the first word of the vector and stored in a temporary location. If x is less than 1, zero is stored in the data vector; if x is equal to or greater than 1 but
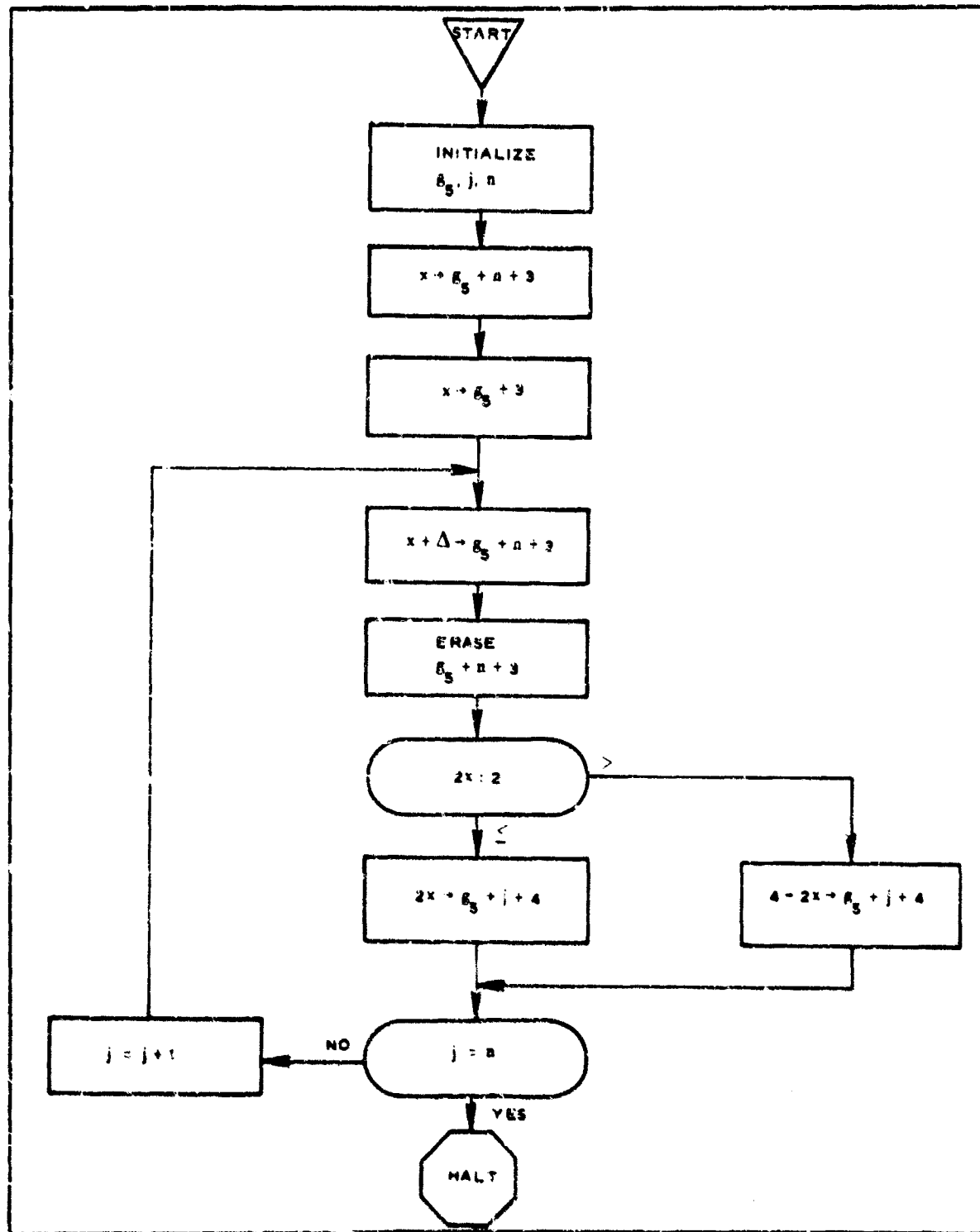
Figure III-5 - Activity Function 5 Flow Chart, Machine I

## TABLE III-8 - ACTIVITY FUNCTION 5 PROGRAM, MACHINE I.

| Item | Instruction | | | | Remarks | Time (μsec) |
|------|-------------|---|---|---|---------|-------------|
| | ENB1 | | | β5 | | 30 |
| | JNP | | | G5 | | |
| G5 | ENB2 | | | ø | | 30 |
| | LDB3 | 1 | | 2 | | |
| | LDA | 1 | | ø | | 30 |
| | STA | 1 | 3 | 3 | | |
| | STA | 1 | | 3 | | 30 |
| | NOP | | | | | |
| G5B | SEH | 1 | 3 | 3 | | 30 |
| | LDA | 1 | 3 | 3 | | |
| | ADD | 1 | | 1 | | 30 |
| | STA | 1 | 3 | 3 | | |
| | ALS | | | 1 | | 30 |
| | INA | | | -2 | | |
| | JNGA | | | G5A | ≤ | 30 |
| | JNZA | | | G5A | = | |
| | INA | | | 2 | ≥ | 30 |
| | STA | 1 | 3 | 4 | | |
| | ENA | | | 4 | | 30 |
| | SEH | 1 | 3 | 4 | | |
| | SUB | 1 | | 4 | | 30 |
| | STA | 1 | 2 | 3 | 4 | |
| G5C | ISK2 | | 3 | ø | | 30 |
| | JNP | | | G5B | | |
| | HLT | | | | | 30 |
| | NOP | | | | | |
| G5A | INA | | | 2 | | 30 |
| | STA | 1 | 2 | 4 | | |
| | JMP | | | G5C | | 30 |
| | NOP | | | | | |

## TABLE III-9 - ACTIVITY FUNCTIONS 5 AND 6

## DATA VECTOR FORMATS, MACHINE I

| Function $g_5(x)$ | | Function $g_6(x)$ | |
|---|---|---|---|
| Address | Data | Address | Data |
| $g_5$ | $x$ | $g_6$ | $x$ |
| $g_5 + 1$ | $\Delta$ | $g_6 + 1$ | $\Delta$ |
| $g_5 + 2$ | $n$ | $g_6 + 2$ | $n$ |
| $g_5 + 3$ | $x_0$ | $g_6 + 3$ | $x_0$ |
| $g_5 + 4$ | $x_1$ | $g_6 + 4$ | $x_1$ |
| $g_5 + 5$ | $x_2$ | $g_6 + 5$ | $x_2$ |
| $g_5 + 6$ | $x_3$ | $g_6 + 6$ | $x_3$ |
| $g_5 + 7$ | $x_4$ | $g_6 + 7$ | $x_4$ |
| $g_5 + 8$ | $x_5$ | $g_6 + 8$ | $x_5$ |
| $g_5 + 9$ | $x_6$ | $g_6 + 9$ | $x_6$ |
| $g_5 + 10$ | $x_7$ | $g_6 + 10$ | $x_7$ |
| $g_5 + 11$ | $x_8$ | $g_6 + 11$ | $x_8$ |
| $g_5 + 12$ | $x_9$ | $g_6 + 12$ | $x_9$ |
| $g_5 + 13$ | $x_{10}$ | $g_6 + 13$ | $x_{10}$ |
| $g_5 + 14$ | $x_{11}$ | $g_6 + 14$ | $x_{11}$ |
| $g_5 + 15$ | $x_{12}$ | $g_6 + 15$ | $x_{12}$ |
| $g_5 + 16$ | $x_{13}$ | $g_6 + 16$ | $x_{13}$ |
| $g_5 + 17$ | $x_{14}$ | $g_6 + 17$ | $x_{14}$ |
| $g_5 + 18$ | $x_{15}$ | $g_6 + 18$ | $x_{15}$ |
| $g_5 + 19$ | $x_{16}$ | $g_6 + 19$ | $x_{16}$ |
| $g_5 + 20$ | $x_{17}$ | $g_6 + 20$ | $x_{17}$ |
| $g_5 + 21$ | $x_{18}$ | $g_6 + 21$ | $x_{18}$ |
| $g_5 + 22$ | $x_{19}$ | $g_6 + 22$ | $x_{19}$ |
| $g_5 + 23$ | $x_{20}$ | $g_6 + 23$ | $x_{20}$ |
| $g_5 + 24$ | TEMP | $g_6 + 24$ | TEMP |
| $g_5 + 25$ | TEMP + 1 | $g_6 + 25$ | TEMP + 1 |
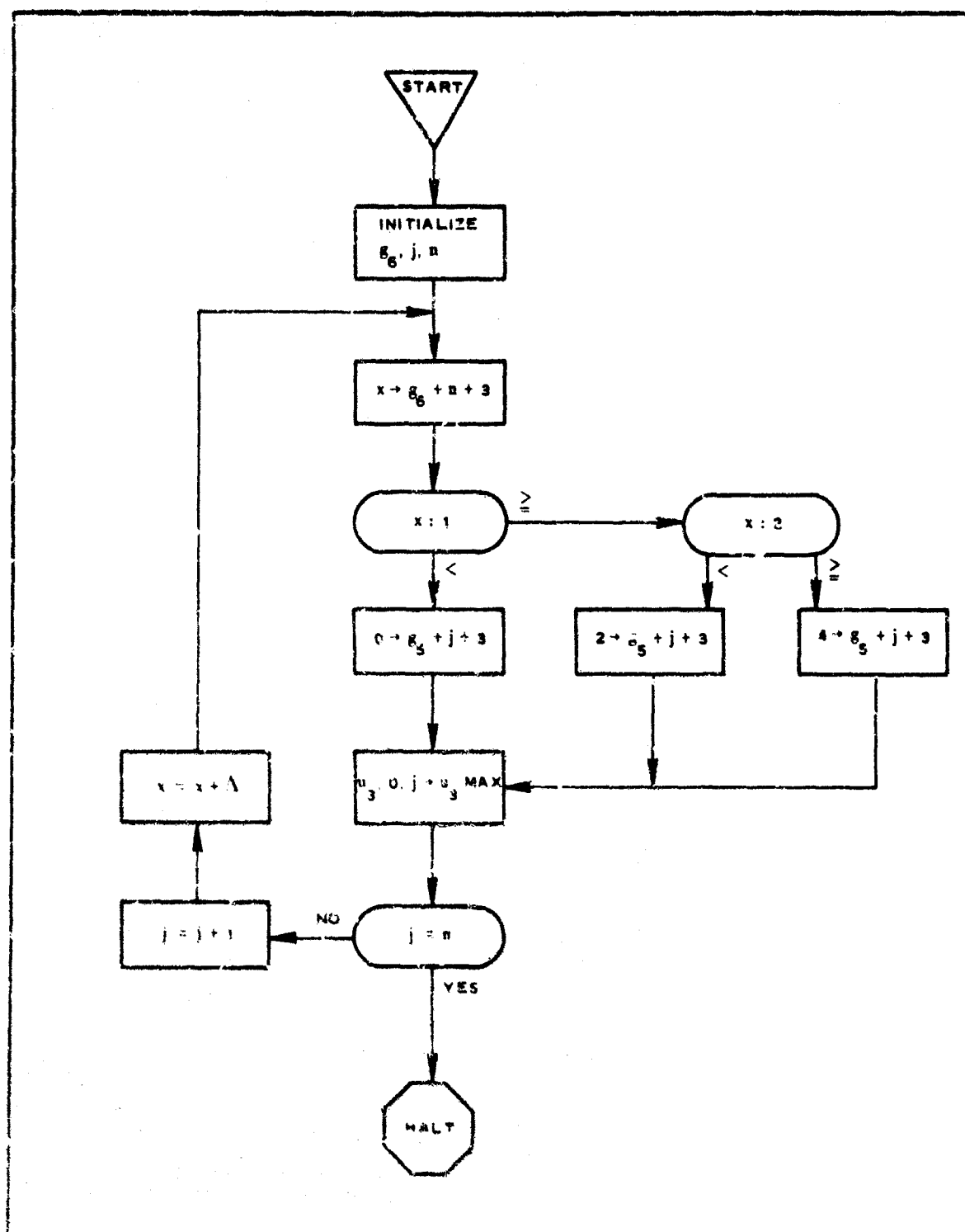
Figure III-6 - Activity Function 6 Flow Chart, Machine I

## TABLE III-10 - ACTIVITY FUNCTION 6 PROGRAM, MACHINE I

| Item | Instruction | | | | Remarks | Time (μsec) |
|------|------|------|------|------|---------|-------------|
|      | ENB1 |   |   | $g_6$ |         | 30 |
|      | JMP  |   |   | G6    |         |    |
| G6   | ENB2 |   |   | $\emptyset$ |   | 30 |
|      | LDB3 | 1 |   | 2     | n       |    |
|      | LDA  | 1 |   | $\emptyset$ |   | 30 |
|      | JMP  |   |   | G6E   |         |    |
| G6D  | LDA  | 1 | 3 | 3     |         | 30 |
|      | ADD  | 1 |   | 1     |         |    |
| G6E  | STA  | 1 | 3 | 3     |         | 30 |
|      | COM  |   |   | 1     |         |    |
|      | JMP  |   |   | G6A   |         | 30 |
|      | LDA  |   |   | $\emptyset$ |   |    |
| G6B  | STA  | 1 | 2 | 3     |         | 30 |
|      | LDA  |   |   | G6M   |         |    |
|      | INAL |   | 2 | 3     |         | 30 |
|      | STA  |   | 2 | MTEM2 |         |    |
|      | LDA  |   |   | GIM + 1 | LDI___ , JMP MAX | 30 |
|      | BGN  |   |   | MTEM + 2 |      |    |
|      | ISK2 |   | 3 | $\emptyset$ |   | 30 |
|      | JMP  |   |   | G6D   |         |    |
|      | HLT  |   |   |       |         | 30 |
|      | NOF  |   |   |       |         |    |
| G6A  | COM  |   |   | 2     |         | 30 |
|      | JMP  |   |   | G6C   |         |    |
|      | ENA  |   |   | 2     |         | 30 |
|      | JMP  |   |   | G6B   |         |    |
| G6C  | ENA  |   |   | 4     |         | 30 |
|      | JMP  |   |   | G6B   |         |    |
|      | $u_3$ |  |   | 0     |         |    |
|      | 0    |   |   | 1     |         |    |

less than 2, then 2 is stored in the data vector; and if x is equal to or greater than 2, then 4 is stored in the vector.

For each value stored in the data vector, a processor is started on the $u_3$ maximization program. The resource for which the value of the activity function has just been calculated is transferred to the $u_3$ maximization program. This enables $u_3$ to begin the maximization process to determine what allocation of this resource will result in the maximum return.

## g. Maximization Function

The flow chart for the maximization function is shown in Figure III-7; the program, in Table III-11; and the data vector format, in Table III-12.

The maximization routine is started by storing an LDI___, JMP MAX instruction in the jump table. The address of the LDI instruction is the address where the indices to be transferred are stored. These addresses - namely, MTEM + B3, MTEM 1 + B3, MTEM 2 + B3, MTEM 3 + B3, and MTEM 4 + B3 - contain, respectively, the indices for execution of the maximization for function $u_1$, $u_2$, $u_3$, $u_4$ and $u_5$. Index register 1 is used to hold address $u_1$, $u_2$, $u_3$, $u_4$, or $u_5$; index register 2 is initially zero; and index register 3 carries j, which indicates the maximum value of resource for which returns are currently available. As the function returns are being calculated, processors are being started with the index information in the MTEM tables.

Index register 4 is loaded with the address located in the second word of the data vector whose address is in index register 1. The contents of index register 1 can be $u_1$, $u_2$, $u_3$, $u_4$, or $u_5$. Correspondingly, word 1 of these data vectors contain the address of the first return in data vectors $g_1$, $g_3$, $g_5$, $u_1$, or $u_4$. Word 2 of each u vector contains the address of the first return in data vectors $g_2$, $g_4$ + 15, $g_6$, $u_3$, or $u_2$. Index register 5 is loaded with one of these corresponding addresses.

The contents of the address are the sum of the contents of index registers 3 and 5. If this address is present in the memory, the contents are loaded into A. If B5 = $g_2$ and B3 = 5 = j + 3, then the return from function $g_2$

-76-
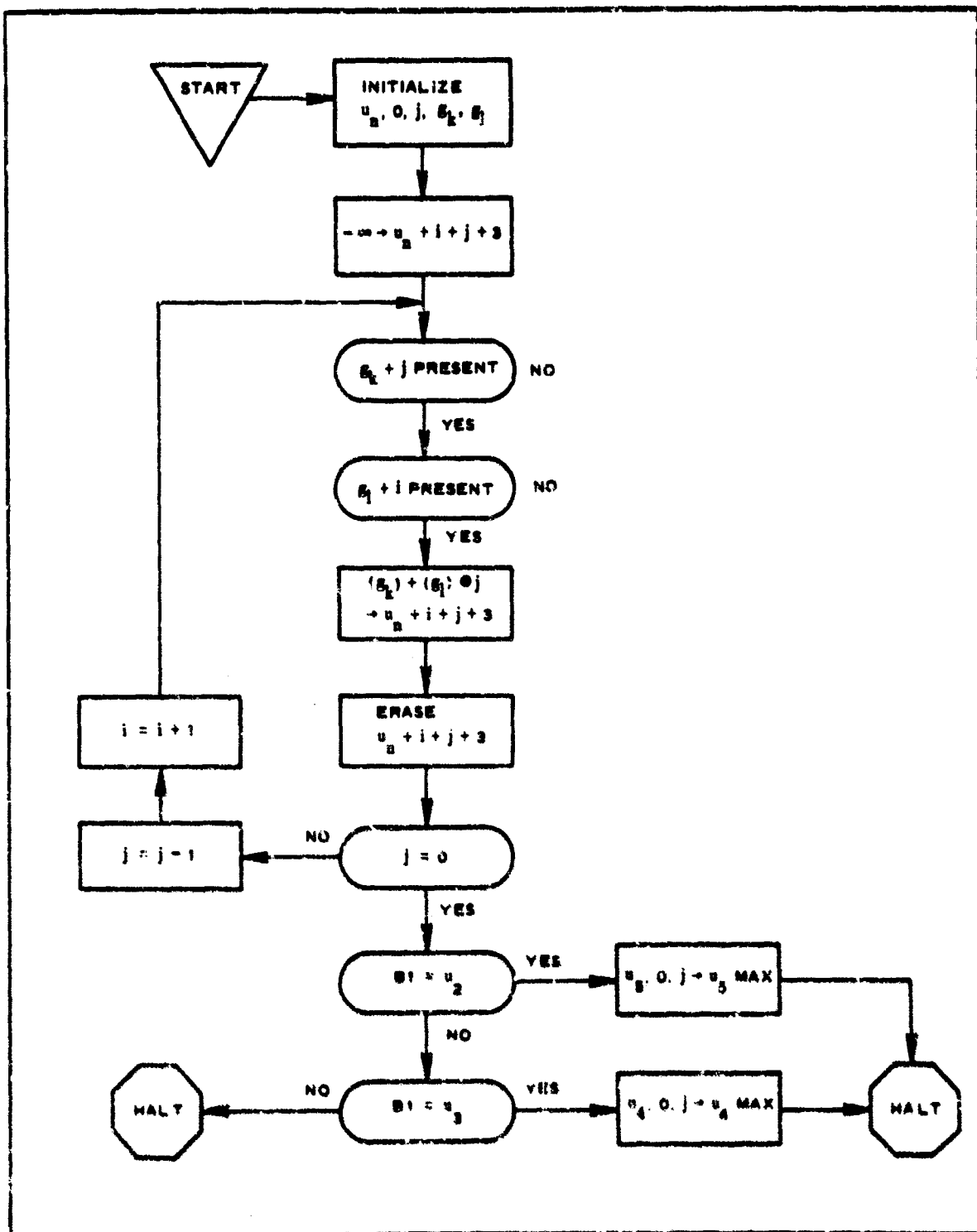
Figure III-7 - Maximisation Function Flow Chart, Machine I

## TABLE III-11 - MAXIMIZATION FUNCTION PROGRAM, MACHINE I

| Item | Instruction | | | | | | Remarks | Time ($\mu$sec) |
|------|-------------|---|---|---|---|---|---------|-----------------|
| | LDI | | | | | . . . | | 30 |
| | JMP | | | | | MAX | | |
| MAX | LDB4 | 1 | | | | 1 | Inputs | 30 |
| | LDB5 | 1 | | | | 2 | B1, B2, B3 | |
| | LDA | | | | | $-\infty$ | $u_1$, 0, j | 30 |
| | STA | 1 | 2 | 3 | | 3 | $u_2$, 0, j | |
| MAXA | LDA | | | 3 | 5 | $\emptyset$ | $u_3$, 0, j | 30 |
| | NPJ | | | | | MAXA | $u_4$, 0, j | |
| MAXC | ADD | | 2 | | 4 | $\emptyset$ | $u_5$, 0, j | 30 |
| | NPJ | | | | | MAXC | | |
| | AND | | | | | LS24 | | 30 |
| | INA | | | 3 | | $\emptyset$ | | |
| | STA | 1 | 2 | 3 | | 3 | | 30 |
| | SEH | 1 | 2 | 3 | | 3 | | |
| | ADD | 1 | 2 | 3 | | 3 | | 30 |
| | BJP3 | | | | | MAXB | | |
| | ENB3 | | 2 | | | $\emptyset$ | | 30 |
| | ENA | | | | | $u_2$ | | |
| | INA | 1 | | | | $\emptyset$ | | 30 |
| | JNZA | | | | | MAXE | | |
| | JMP | | | | | MAXD | DOu$_5$ | 30 |
| | NOP | | | | | | | |
| MAXE | ENA | | | | | $u_3$ | | 30 |
| | INA | 1 | | | | $\emptyset$ | | |
| | JNZA | | | | | HLT | | 30 |
| | LDB1 | | | | | $u_4$ | DOu$_4$ | |
| | ENB2 | | | | | $\emptyset$ | | 30 |

TABLE III-11 - MAXIMIZATION FUNCTION PROGRAM, MACHINE I(Continued)

| Item | Instruction | | Remarks | | Time (µsec) |
|------|------|------|------|------|------|
| | STI | 3 | MTEM3 | | |
| | LDA | | GIM + 1 | LDI MTEM3 + B3, JMP MAX | 30 |
| | BGN | 3 | MTEM3 | | |
| | HLT | | | | 30 |
| | NOP | | | | |
| MAXD | LDB1 | | $u_5$ | | 30 |
| | ENB2 | $\phi$ | | | |
| | STI | 3 | MTEM4 | | 30 |
| | LDA | | GIM + 1 | LDI MTEM4 + B3, JMP MAX | |
| | BGN | 3 | MTEM4 | | 30 |
| | HLT | | | | |
| MAXB | INB2 | | 1 | | 30 |
| | JMP | | MAXA | | |
| ML | LDI | | | | |
| | JMP | | MAX | | |
| MTEM | $u_1$ | | 0 | | |
| | 0 | | j | | |
| MTEM1 | $u_2$ | | 0 | | |
| | 0 | | j | | |
| MTEM2 | $u_3$ | | 0 | | |
| | 0 | | j | | |
| MTEM3 | $u_4$ | | 0 | | |
| | 0 | | j | | |
| MTEM4 | $u_5$ | | 0 | | |
| | 0 | | j | | |

### TABLE III-12 - MAXIMIZATION FUNCTION DATA VECTOR FORMAT,
### MACHINE I

| Function $u_1$ | Function $u_2$ | Function $u_3$ | Function $u_4$ | Function $u_5$ |
|---|---|---|---|---|
| $u_1$ | $u_2$ | $u_3$ | $u$ | $u_5$ |
| $g_1$ | $g_3$ | $g_5$ | $u_1$ | $u_4$ |
| $g_2$ | $g_{4+15}$ | $g_6$ | $u_3$ | $u_2$ |
| $u_{1(0)};\ y_{1(0)}$ | $u_{2(0)};\ y_{2(0)}$ | $u_{3(0)};\ y_{2(0)}$ | $u_{4(0)};\ y_{4(0)}$ | $u_{5(0)};\ y_{5(0)}$ |
| $u_{1(1)};\ y_{1(1)}$ | $u_{2(1)};\ y_{2(1)}$ | $u_{3(1)};\ y_{2(1)}$ | $u_{4(1)};\ y_{4(1)}$ | $u_{5(1)};\ y_{5(1)}$ |
| $u_{1(2)};\ y_{1(2)}$ | $u_{2(2)};\ y_{2(2)}$ | $u_{3(2)};\ y_{2(2)}$ | $u_{4(2)};\ y_{4(2)}$ | $u_{5(2)};\ y_{5(2)}$ |
| $u_{1(3)};\ y_{1(3)}$ | $u_{2(3)};\ y_{2(3)}$ | $u_{3(3)};\ y_{2(3)}$ | $u_{4(3)};\ y_{4(3)}$ | $u_{5(3)};\ y_{5(2)}$ |
| $u_{1(4)};\ y_{1(4)}$ | $u_{2(4)};\ y_{2(4)}$ | $u_{3(4)};\ y_{2(4)}$ | $u_{4(4)};\ y_{4(4)}$ | $u_{5(4)};\ y_{5(4)}$ |
| $u_{1(5)};\ y_{1(5)}$ | $u_{2(5)};\ y_{2(5)}$ | $u_{3(5)};\ y_{2(5)}$ | $u_{4(5)};\ y_{4(5)}$ | $u_{5(5)};\ y_{5(5)}$ |
| $u_{1(6)};\ y_{1(6)}$ | $u_{2(6)};\ y_{2(6)}$ | $u_{3(6)};\ y_{2(6)}$ | $u_{4(6)};\ y_{4(6)}$ | $u_{5(6)};\ y_{5(6)}$ |
| $u_{1(7)};\ y_{1(7)}$ | $u_{2(7)};\ y_{2(7)}$ | $u_{3(7)};\ y_{2(7)}$ | $u_{4(7)};\ y_{4(7)}$ | $u_{5(7)};\ y_{5(7)}$ |
| $u_{1(8)};\ y_{1(8)}$ | $u_{2(8)};\ y_{2(8)}$ | $u_{3(8)};\ y_{2(8)}$ | $u_{4(8)};\ y_{4(8)}$ | $u_{5(8)};\ y_{5(8)}$ |
| $u_{1(9)};\ y_{1(9)}$ | $u_{2(9)};\ y_{2(9)}$ | $u_{3(9)};\ y_{2(9)}$ | $u_{4(9)};\ y_{4(9)}$ | $u_{5(9)};\ y_{5(9)}$ |
| $u_{1(10)};\ y_{1(10)}$ | $u_{2(10)};\ y_{2(10)}$ | $u_{3(10)};\ y_{2(10)}$ | $u_{4(10)};\ y_{4(10)}$ | $u_{5(10)};\ y_{5(10)}$ |
| $u_{1(11)};\ y_{1(11)}$ | $u_{2(11)};\ y_{2(11)}$ | $u_{3(11)};\ y_{2(11)}$ | $u_{4(11)};\ y_{4(11)}$ | $u_{5(11)};\ y_{5(11)}$ |
| $u_{1(12)};\ y_{1(12)}$ | $u_{2(12)};\ y_{2(12)}$ | $u_{3(12)};\ y_{2(12)}$ | $u_{4(12)};\ y_{4(12)}$ | $u_{5(12)};\ y_{5(12)}$ |
| $u_{1(13)};\ y_{1(13)}$ | $u_{2(13)};\ y_{2(13)}$ | $u_{3(13)};\ y_{2(13)}$ | $u_{4(13)};\ y_{4(13)}$ | $u_{5(13)};\ y_{5(13)}$ |
| $u_{1(14)};\ y_{1(14)}$ | $u_{2(14)};\ y_{2(14)}$ | $u_{3(14)};\ y_{2(14)}$ | $u_{4(14)};\ y_{4(14)}$ | $u_{5(14)};\ y_{5(14)}$ |
| $u_{1(15)};\ y_{1(15)}$ | $u_{2(15)};\ y_{2(15)}$ | $u_{3(15)};\ y_{2(15)}$ | $u_{4(15)};\ y_{4(15)}$ | $u_{5(15)};\ y_{5(15)}$ |
| $u_{1(16)};\ y_{1(16)}$ | $u_{2(16)};\ y_{2(16)}$ | $u_{3(16)};\ y_{2(16)}$ | $u_{4(16)};\ y_{4(16)}$ | $u_{5(16)};\ y_{5(16)}$ |
| $u_{1(17)};\ y_{1(17)}$ | $u_{2(17)};\ y_{2(17)}$ | $u_{3(17)};\ y_{2(17)}$ | $u_{4(17)};\ y_{4(17)}$ | $u_{5(17)};\ y_{5(17)}$ |
| $u_{1(18)};\ y_{1(18)}$ | $u_{2(18)};\ y_{2(18)}$ | $u_{3(18)};\ y_{2(18)}$ | $u_{4(18)};\ y_{4(18)}$ | $u_{5(18)};\ y_{5(18)}$ |
| $u_{1(19)};\ y_{1(19)}$ | $u_{2(19)};\ y_{2(19)}$ | $u_{3(19)};\ y_{2(19)}$ | $u_{4(19)};\ y_{4(19)}$ | $u_{5(19)};\ y_{5(19)}$ |
| $u_{1(20)};\ y_{1(20)}$ | $u_{2(20)};\ y_{2(20)}$ | $u_{3(20)};\ y_{2(20)}$ | $u_{4(20)};\ y_{4(20)}$ | $u_{5(20)};\ y_{5(20)}$ |

for a resource allocation of 0.2 is loaded into A. Similarly the return for a zero resource assignment to function $g_1$ is obtained via B4 and B2 and added to the previous contents of the accumulator. The least significant 24 bits of the sum are replaced by (B3) = j. The contents of A, bits 24 to 71, are now the sum of the returns from activity function $g_2$ and $g_1$ for resource assignments of 0.2 to $g_2$ and 0 to $g_1$. The J, representing the assignment of 0.2 units to $g_2$, is stored in the least significant 24 bits; A contents are stored in data vectors $u_1$ at the address that is the sum of the contents of index registers 2 and 3 plus the contents of the address position of the instruction; or B1 + B2 + B3 + 3 = $u_1$ + 0 + 2 + 3.

For a resource of 0.2, there are now 2 more possible allocation sets: 0.1 to $g_2$ and 0.1 to $g_1$, and 0 to $g_2$ and 0.2 to $g_1$. Returns for these two assignments are found as indicated earlier, and they are stored in words with the same address (name). For example, the sum of the $g_2$ and $g_1$ returns for a resource allocation of 0.2 to $g_2$ and 0 to $g_1$ is stored in a word with an address equal to B1 + B2 + B3 + 3, or $u_1$ + 0 + 2 + 3. The returns for other permissible combinations of resource allocation - such as 0.1 to $g_2$ and 0.1 to $g_1$; and 0 to $g_2$ and 0.2 to $g_1$ - are stored in memory with the same name. The name is derived from the contents of index register 1 and the quantity of resource that is to be allocated. Since a large negative value was stored with this name upon entry to the maximization routine, the first larger entry stored with this name will be located above the large negative value. The next instruction after the store (STA 123 3) is a single erase high (SEH 123 3) followed by a fetch type instruction. The result of this execution is to erase the word where the name equal to the contents of the sum of B1, B2, B3, and 3 and where magnitude is smallest. In this case, after one storage instruction two words are in memory have the same name:

| Name | Data |
|------|------|
| $u_1$ + 5 | some + value |
| $u_1$ + 5 | large - value |

After the SEH instruction, the word with the large negative value is erased. With this sequence of instructions, successive values of a function could be stored in the memory; and after a machine cycle, one word of the data vector could be erased, always leaving the largest value in the vector at the top of the vector. Index register 3, in combination with index register 2, determines the combinations of allowable resource allocations. Initially, index 3 has the maximum current resource, and index 2 has the minimum compatible resource to be allocated. In the process of finding the maximum return for a given resource assignment, index 3 is decremented and index 2 is incremented; for each combination, the appropriate activity returns are found, added together, and stored in the data vector where the smallest element of the vector is erased and the largest element of the vector retained.

Each input to the maximization function results in the formation of a segment of one of the maximized return vectors. The transfer of index data $u_1$, 0, and j to a processing unit results in j + 1 pieces of data being stored in vector $u_1$, all with names $u_1$ + j. When the calculation of the j + 1 pieces of data is completed, only the largest is retained, and it is located in jth position of the $u_1$ vector.

In this example problem, there are 105 possible combinations of inputs to this single routine. This means that a possible 105 processing units are operating on one program. Five vectors are to be generated - $u_1$, $u_2$, $u_3$, $u_4$, $u_5$ - and j can range between 0 and 20 in steps of 1.

There is even more parallelism in this section of the problem. Each possible resource combination could have been assigned to an arithmetic unit. In this sample problem, there are 21 possible resource allocations to the 5 vectors for a total of 1155 possible combinations of resource to be maximized. Each combination could have been assigned to a processor unit. This approach, however, would entail more calculation to specify the input values of the indices. In the present method, only the maximum value, j, of the resource is transferred, and a loop is executed j + 1 times.

The remainder of this program sets up the indices to start processing units calculating the $u_4$ and $u_5$ vectors. Whenever work is being done on vectors $u_2$ or $u_3$ and the largest value has been found for some resource, j, then j and the $u_4$ or $u_5$ indices are transferred to a new processing unit to start work on the $u_4$ or the $u_5$ vector.

### h. Lookup Function

The flow chart for the lookup function is shown in Figure III-8; and the program, in Table III-13.

The input to the lookup function is the quantity of resource x to be allocated to the 6 activity functions. The resource is used in combination with the name of a maximized data vector to search the vector for the recommended resource assignment. When the recommended assignment is found, the remainder is assigned to the next maximized vector; and so on.

The search word is made up of the NHA instruction and the contents of B1, which in turn is equal to the name of the data vector and the quantity of resource to be allocated. This search word is used to find the element of the vector that has the recommended assignment for that quantity of resource. The recommended assignments are stored in the 0 list of Table III-13.

## 3. RESULTS

### a. Activity Functions 1 and 2

The timing charts for the $g_1(x)$ and the $g_2(x)$ activity functions are shown in Figures III-9 and III-10, respectively. Since these functions were relatively simple expressions, it was felt that little would be gained in time by attempting to compute their various values in parallel. Hence, one processing unit was assigned to each function, and an iterative program was written that evaluates the function over the range of the argument. Each function turns out a new value of the function about every 180 usec. For each new functional value calculated, the $g_i(x)$
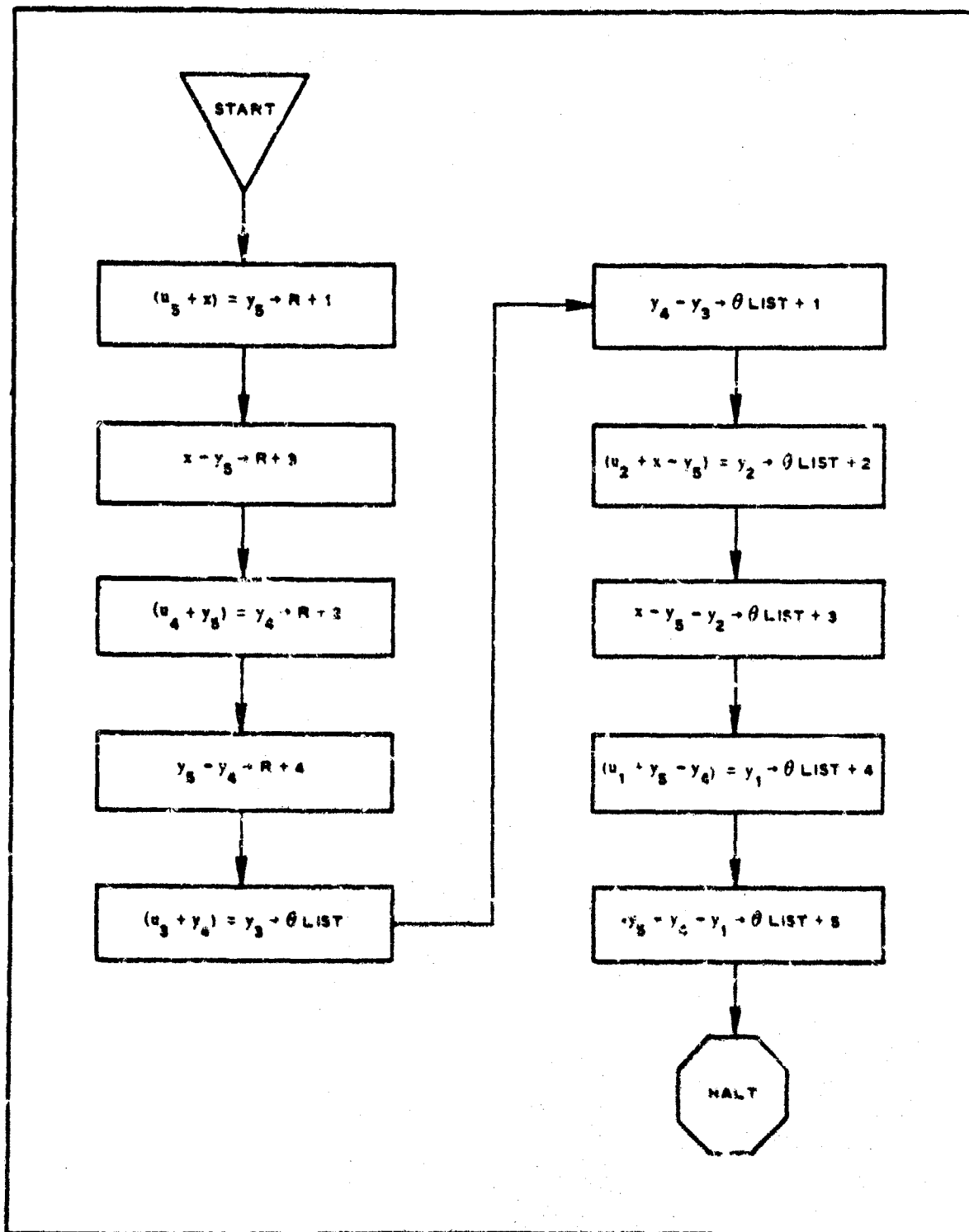
Figure III-8 - Lookup Function Flow Chart, Machine I

## TABLE III-13 - LOOKUP FUNCTION PROGRAM, MACHINE I

| Item | Instruction | | | |
|------|------|------|------|------|
| | JMP | | | LOOKUP |
| LOOKUP | LDB1 | | | IN ADD |
| | INB1 | | | R |
| | NHA | 1 | | ∅ |
| | ENB4 | | | LA |
| | STB4 | | | R + 1 |
| | LDA | | | R |
| | SUB | | | R + 1 |
| | STA | | | R + 3 |
| | INB2 | | | 1 |
| | LDB1 | 2 | | IN ADD |
| | INB1 | | | R + 1 |
| | NHA | 1 | | ∅ |
| | ENB4 | | | LA |
| | STB4 | | | R + 2 |
| | LDA | | | R + 1 |
| | SUB | | | R + 2 |
| | STA | | | R + 4 |
| | ENB3 | | | 2 |
| | ENB5 | | | 1 |
| LUI | INB2 | | | 1 |
| | LDB1 | 2 | | IN ADD |
| | INB1 | | 5 | R + 2 |
| | NHA | 1 | | ∅ |
| | ENB4 | | | LA |
| | STB4 | 3 | | θ list |
| | LDA | | 5 | R + 2 |
| | SUB | 3 | | θ list |
| | STA | 3 | | θ list + 1 |

## TABLE III-13 - LOOKUP FUNCTION PROGRAM, MACHINE I

### (Continued)

| Item | Instruction | |
|------|-------------|---|
| | INB3 | 2 |
| | ISK5 | 3 |
| | JMP | LU1 |
| | HLT | |
| IN ADD | NOP | |
| | $u_5$ | |
| | $u_4$ | |
| | $u_3$ | |
| | $u_2$ | |
| | $u_1$ | |
| R | $x$ | |
| | $y_5$ | |
| | $y_4$ | |
| | $x - y_5$ | |
| | $y_5 - y_4$ | |
| θ LIST | $g_6$ | $y_3$ |
| | $g_5$ | $y_4 - y_3$ |
| | $g_4$ | $y_2$ |
| | $g_3$ | $x - y_5 - y_2$ |
| | $g_2$ | $y_1$ |
| | $g_1$ | $y_5 - y_6 - y_1$ |
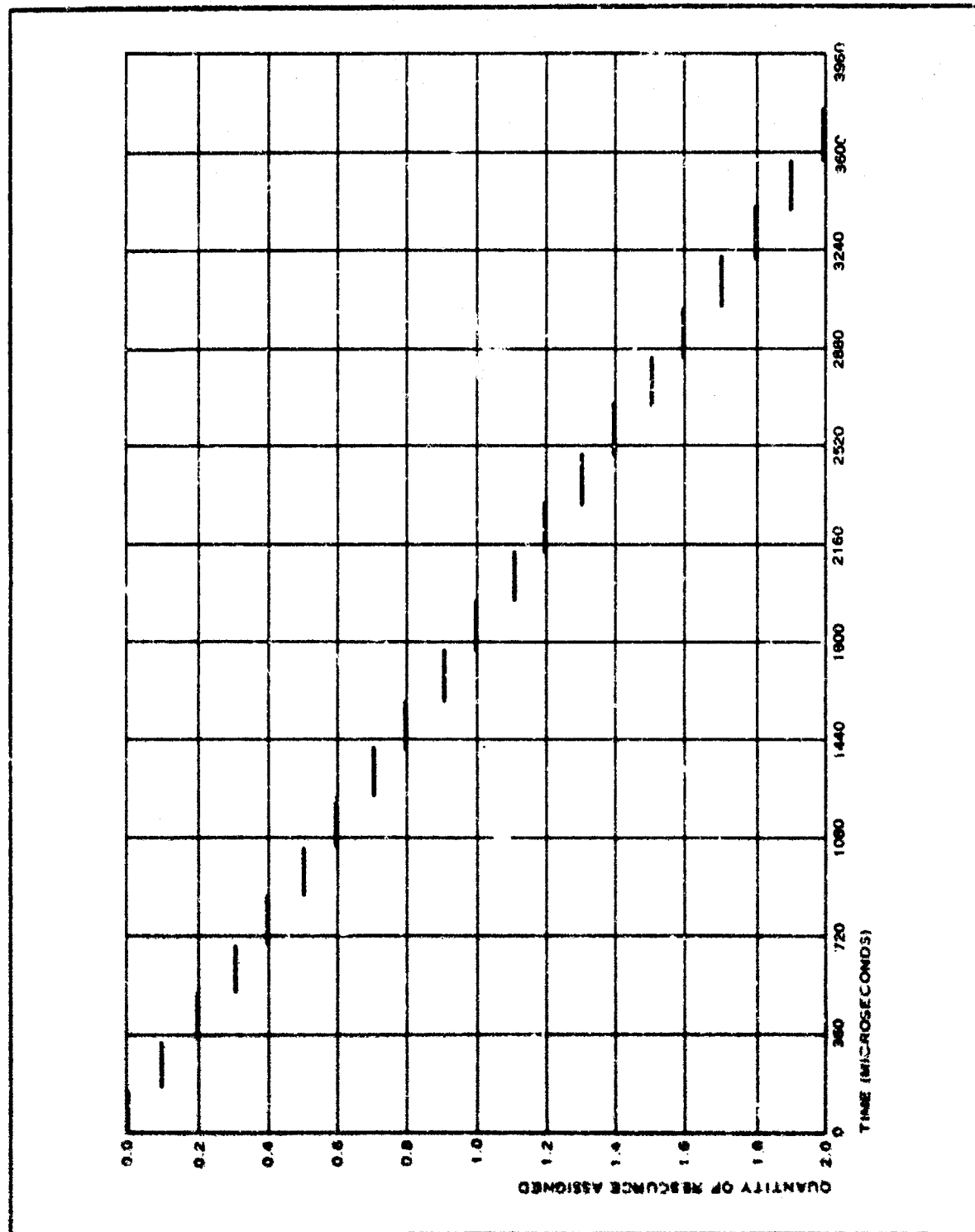
Figure III-9 - Activity Function 1 Timing Chart, Machine I
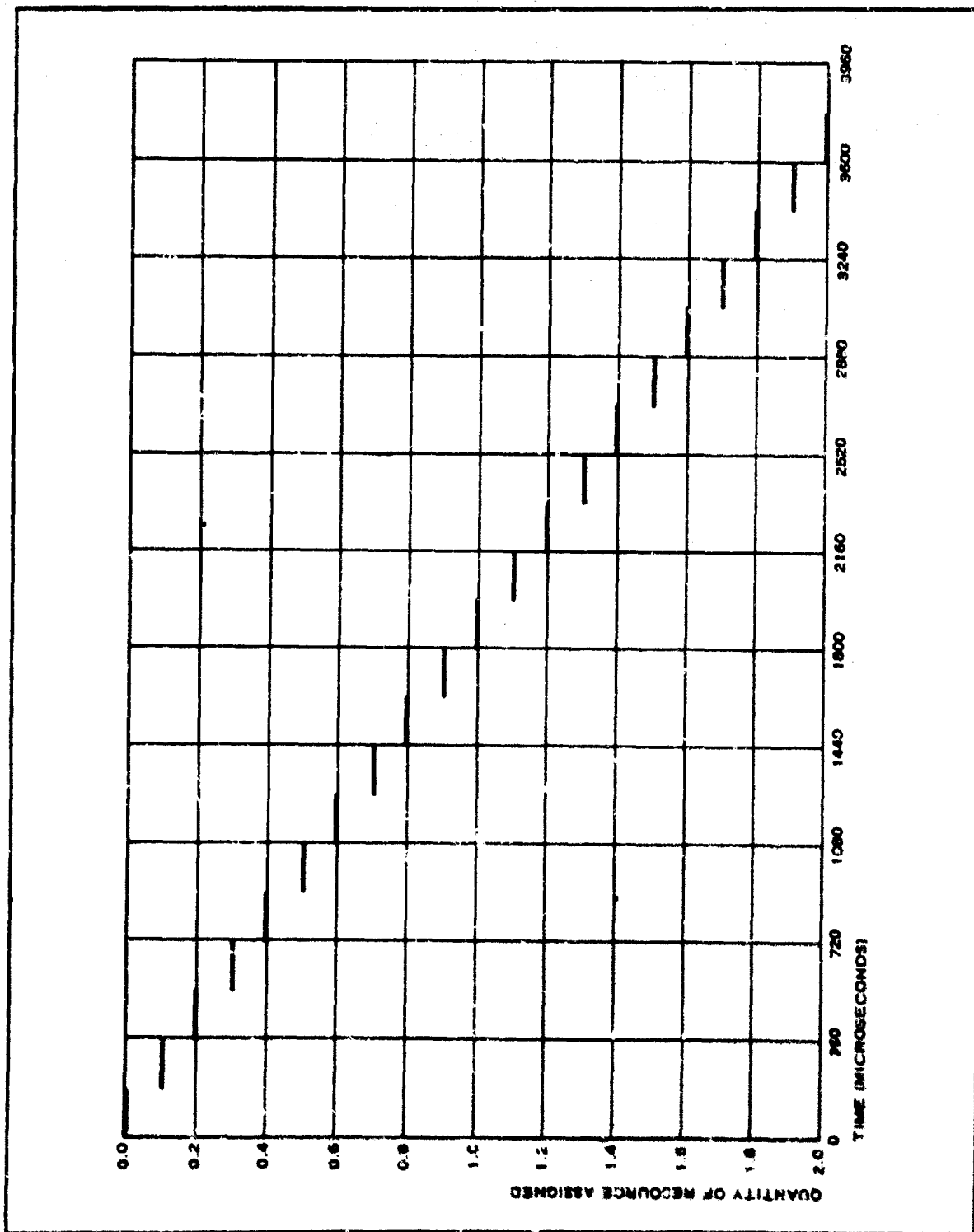
Figure III-10 - Activity Function 2 Timing Chart, Machine I

function generates the start instructions and indices for the $u_1$ maximization program. In this manner maximization continues in parallel but slightly behind the activity function calculation. The $u_1$ maximization program uses as its inputs the functional values of the $g_1(x)$ and $g_2(x)$ programs, which use 1 processor each and calculate in about 3780 µsec.

### b. Activity Function 3

The timing chart for the $g_3(x)$ activity function is shown in Figure III-11. This function displays more parallelism than the $g_1(x)$ and $g_2(x)$ function. For each input to the program, three processors are activated. One processor calculates indices to maintain the treeing operation, one executes the iterative loop, and another tests the exponent of the floating joint input number. The number of processors in use varies from 1 to 35, and total time is about 1950 µsec.

The treeing operation is maintained by calculating from the input indices two more sets of indices, which in turn result in the calculation of four sets of indices. Each level of index calculation results in a number of index sets equal to a power of two. The quantity of data being generated increases exponentially, while the time to generate the data increases linearly.

The approximate minimum and maximum output-data times at the second through eighth levels of indexing are shown in Table III-14; note that the minimum increase is 180 µsec, and the maximum increase 300 µsec, from one level to the next. It is possible to predict the calculation of a large number of square roots; for example, 256 in a time equal to 3090 µsec.

### c. Activity Function 4

The timing chart for the $g_4(x)$ activity function is shown in Figure III-12. This function computes the sines and cosines of 42 angles in about 4300 µsec, using a maximum of 26 processors. Except for initial calculations, four processors operate on each value of the input argument. Two

### TABLE III-14 - ACTIVITY FUNCTION 3 MINIMUM AND MAXIMUM OUTPUT-DATA TIMES, MACHINE I

| Level | Output-data time ($\mu$sec) | | Data elements |
|-------|---------|---------|---------------|
|       | Minimum | Maximum | |
| 2 | 1170 | 1290 | 2 |
| 3 | 1350 | 1590 | 4 |
| 4 | 1530 | 1890 | 8 |
| 5 | 1710 | 2190 | 16 |
| 6 | 1890 | 2490 | 32 |
| 7 | 2070 | 2790 | 64 |
| 8 | 2250 | 3090 | 128 |

processors calculate the sine and cosine of the incremental angle; two more calculate the sine and cosine of the base angle, plus the incremental angle, while index treeing calculations are being performed by the previous processors.

In this routine, the minimum and maximum times between index levels increase at a linear rate; while the quantity of data being generated increases exponentially for a linear increase in time.

The approximate minimum and maximum output-data times at the second through fifth levels of indexing are shown in Table III-15; note that the minimum increase is 510 $\mu$sec, and the maximum increase 690 $\mu$sec, from one level to the next. It is possible to generate 256 sets of sines and cosines in about 6650 $\mu$sec.

### d. Activity Functions 5 and 6

The timing charts for the $g_5(x)$ and the $g_6(x)$ activity functions are shown in Figures III-13 and III-14, respectively. These functions are similar to the $g_1(x)$ and $g_2(x)$ function in that one processor was assigned to each function because of the low inherent parallelism.
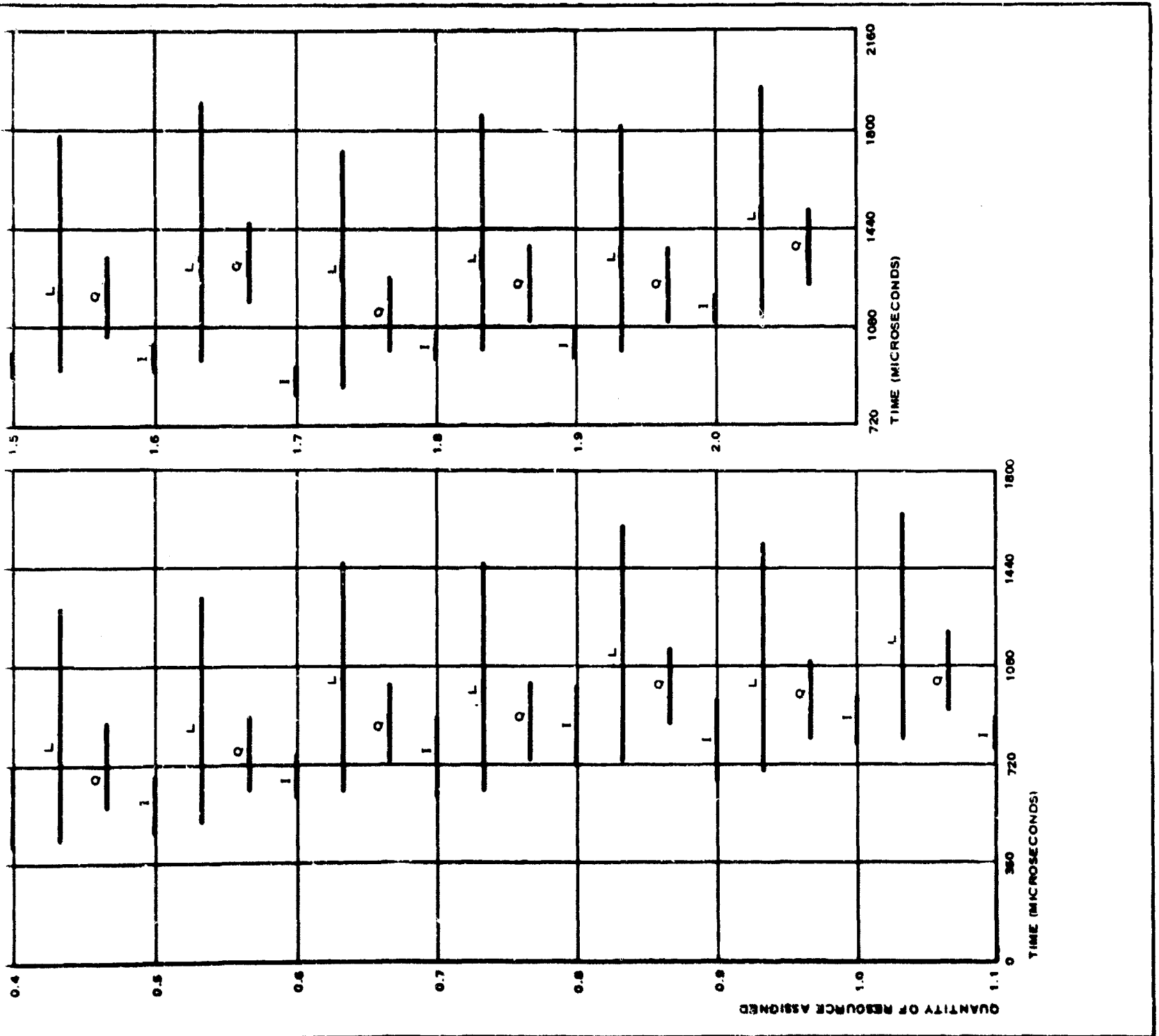
Figure III-11 - Activity Function 3 Timing Chart, Machine I
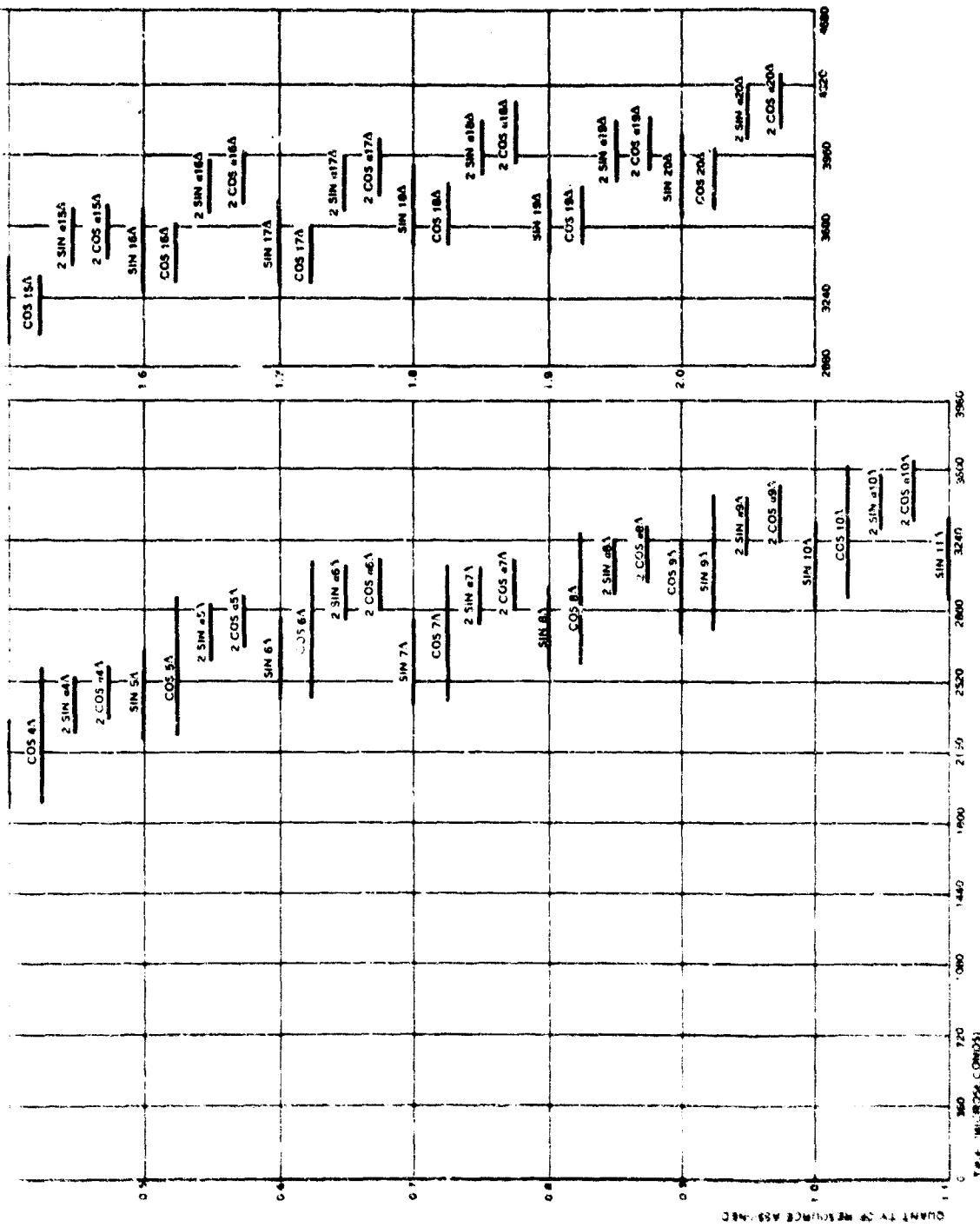
Figure III-12 .

Figure III-12 - Activity Function 4 Timing Chart, Machine I

B

TABLE III-15 - ACTIVITY FUNCTION 4 MINIMUM AND

MAXIMUM OUTPUT-DATA TIMES, MACHINE I

| Level | Output-data time ($\mu$sec) | | Data elements |
|---|---|---|---|
| | Minimum | Maximum | |
| 2 | 2340 | 2500 | 2 |
| 3 | 2850 | 3200 | 4 |
| 4 | 3360 | 3890 | 8 |
| 5 | 3870 | . . . | . . . |

About 450 $\mu$sec are required to generate a new element of the output data vector. The total output vector for $g_5(x)$ is available at 4560 $\mu$sec; and for $g_6(x)$, at 4710 $\mu$sec. In addition, the $g_6(x)$ generates indices to be transferred to the $u_3$ maximization program.

e.  Maximization Function

The data flow diagram for the maximization function is shown in Figure III-15. It should be noted that the inputs to the maximization function are sets of indices that allow generation of the maximized return vectors $u_1$ through $u_5$, referred to here as maximization programs. The timing chart for the $u_1$ maximization program is shown in Figure III-16; for the $u_2$ program, in Figure III-17; for the $u_3$ program, in Figure III-18; for the $u_4$ program, in Figure III-19; and for the $u_5$ program, in Figure III-20.

The $u_1$ program of the maximization function gets its input indices from the $g_1(x)$ activity function. Each input allows the maximization program to compute all possible combinations of returns from $g_1(x)$ and $g_2(x)$ for the given resource to determine what resource allocation will give the maximum return. Each time the $g_1(x)$ activity function generates a set of indices for the maximization program, a processor is started. As the resource to be maximized increases, the time required to maximize
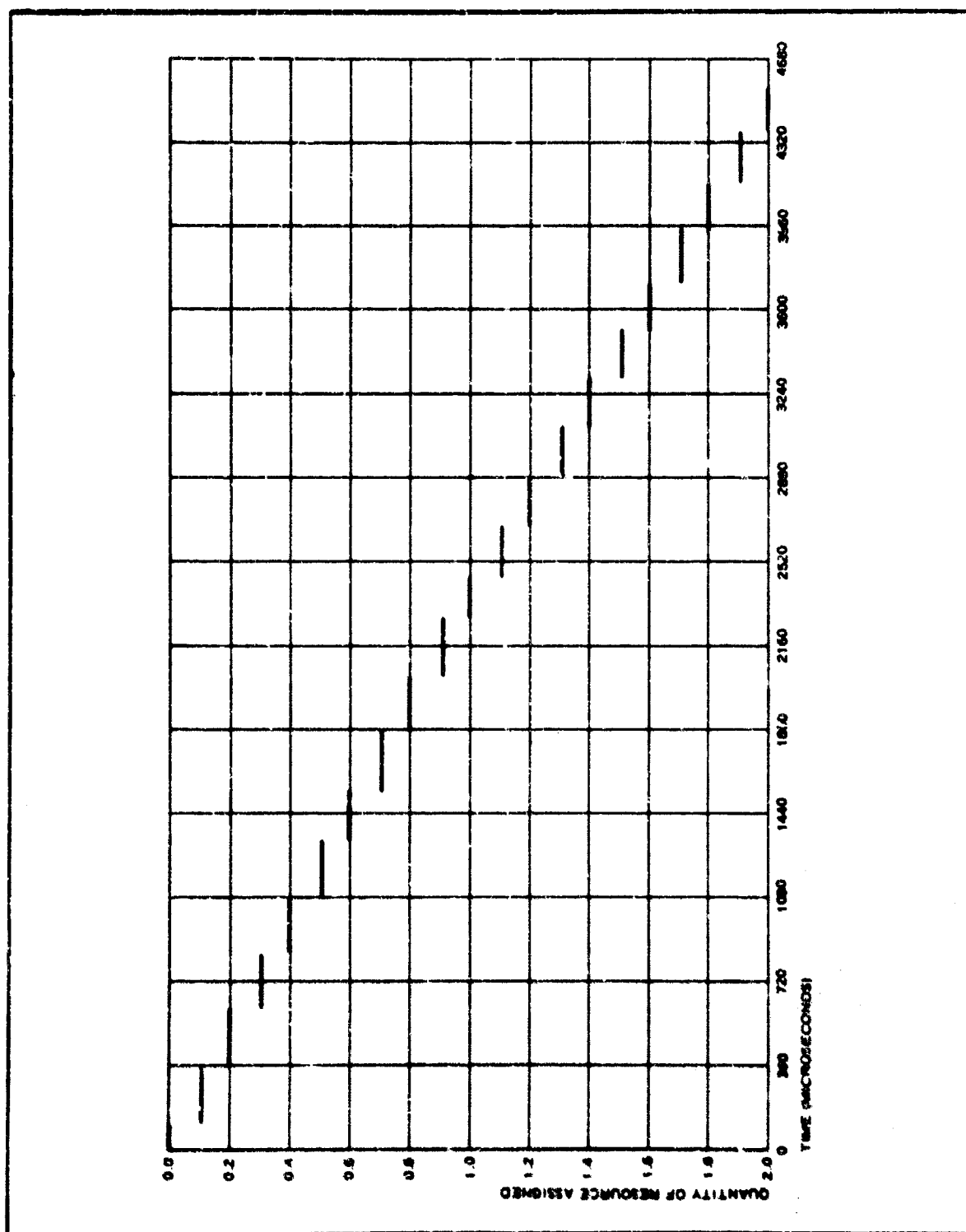
Figure III-13 - Activity Function 5 Timing Chart, Machine I
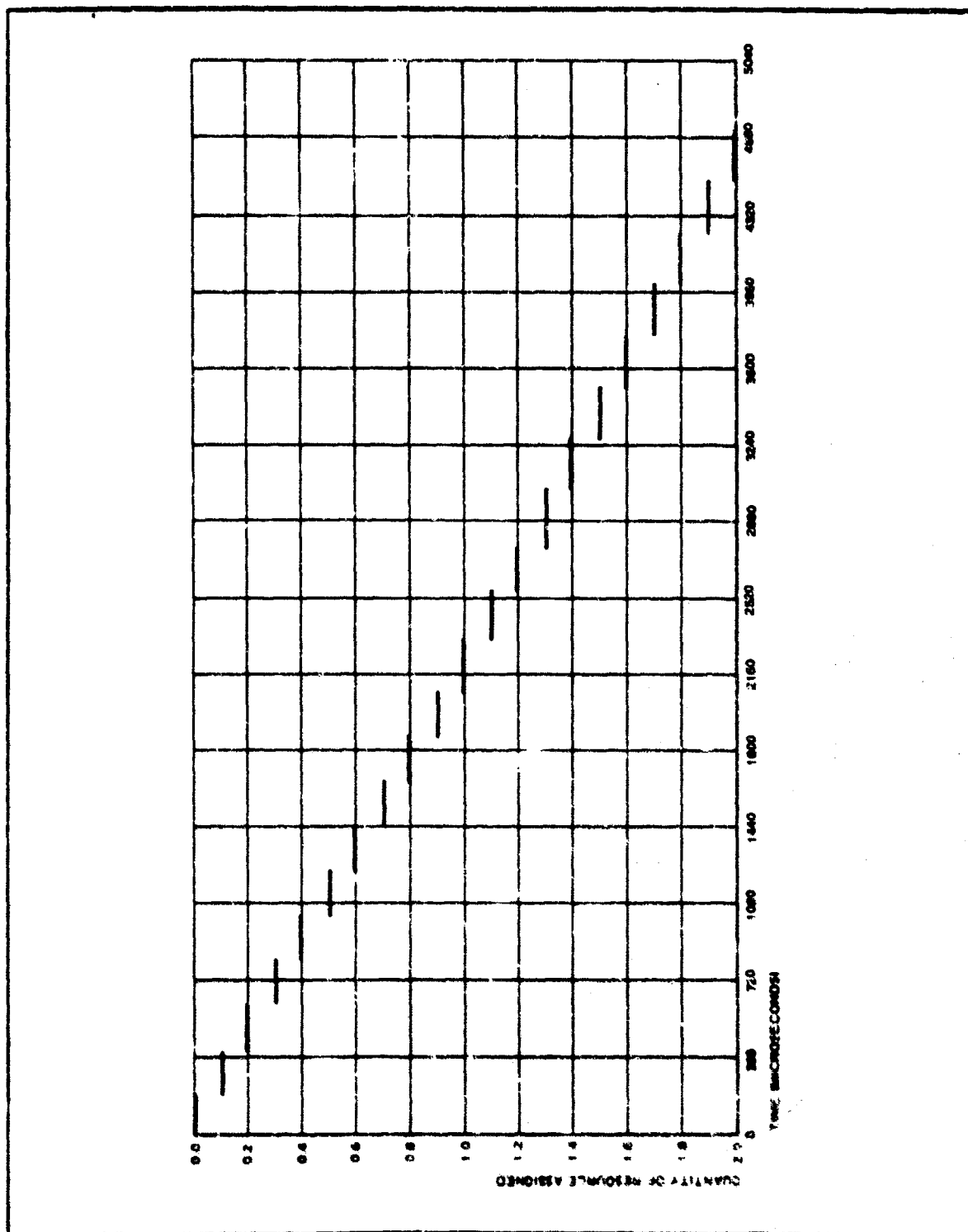
Figure III-14 - Activity Function 6 Timing Chart, Machine I

APPENDIX III
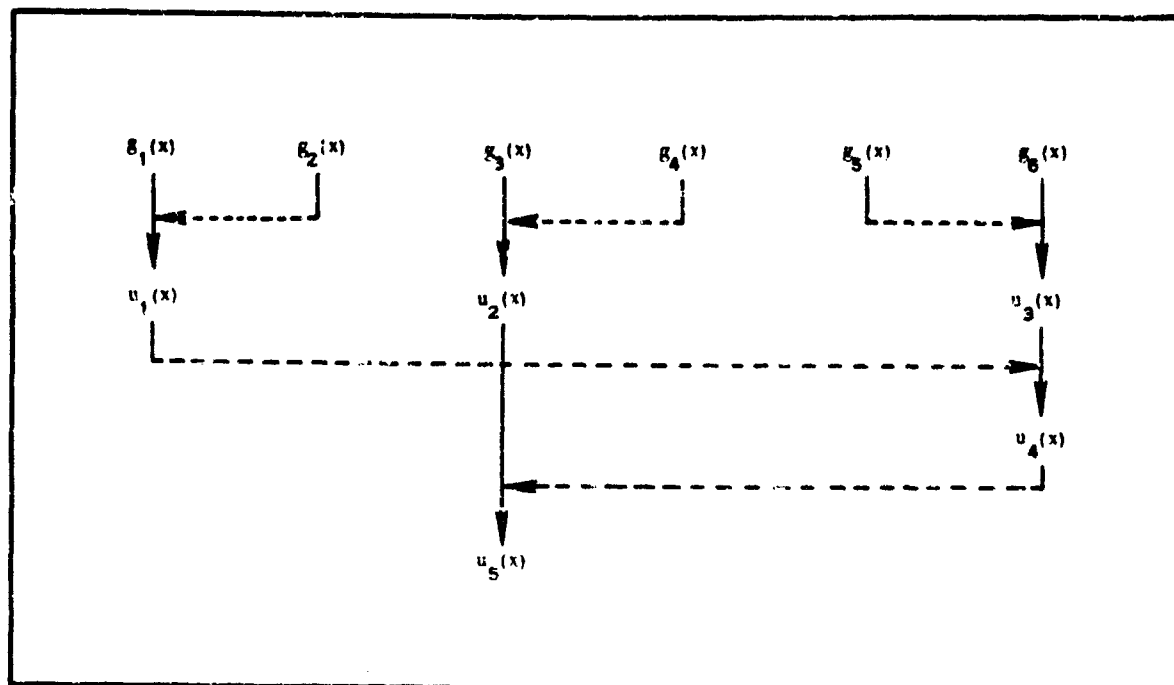


Figure III-15 - Maximization Programs Data Flow Diagram, Machine I

the return for this resource increases; for the $u_1$ program, this time amounts to 7500 $\mu$sec (see Figure III-16).

The $u_2$ and $u_3$ programs acquire their inputs from the $g_3(x)$ and $g_6(x)$ activity functions, respectively. The $u_4$ and $u_5$ programs are started from the $u_3$ and $u_2$ programs, respectively.

The maximization programs are started as soon as data are generated that a program can operate on. The dashed lines on the $u_5$ timing chart (Figure III-20) represent idle (wheel spinning) time where the arithmetic unit is looking for a piece of data yet to be generated. Until the data are generated by the $u_4$ program, the $u_5$ processor is not constructively useful.

As an explanation of the $u_5$ wheel spinning time, the $u_2$ program generates indices to start the $u_5$ program, and there is a disparity between (1) the time the $u_2$ program generates data for the start of $u_5$ and (2) the time that $u_4$ has corresponding data ready for $u_5$. A solution to the

-98-
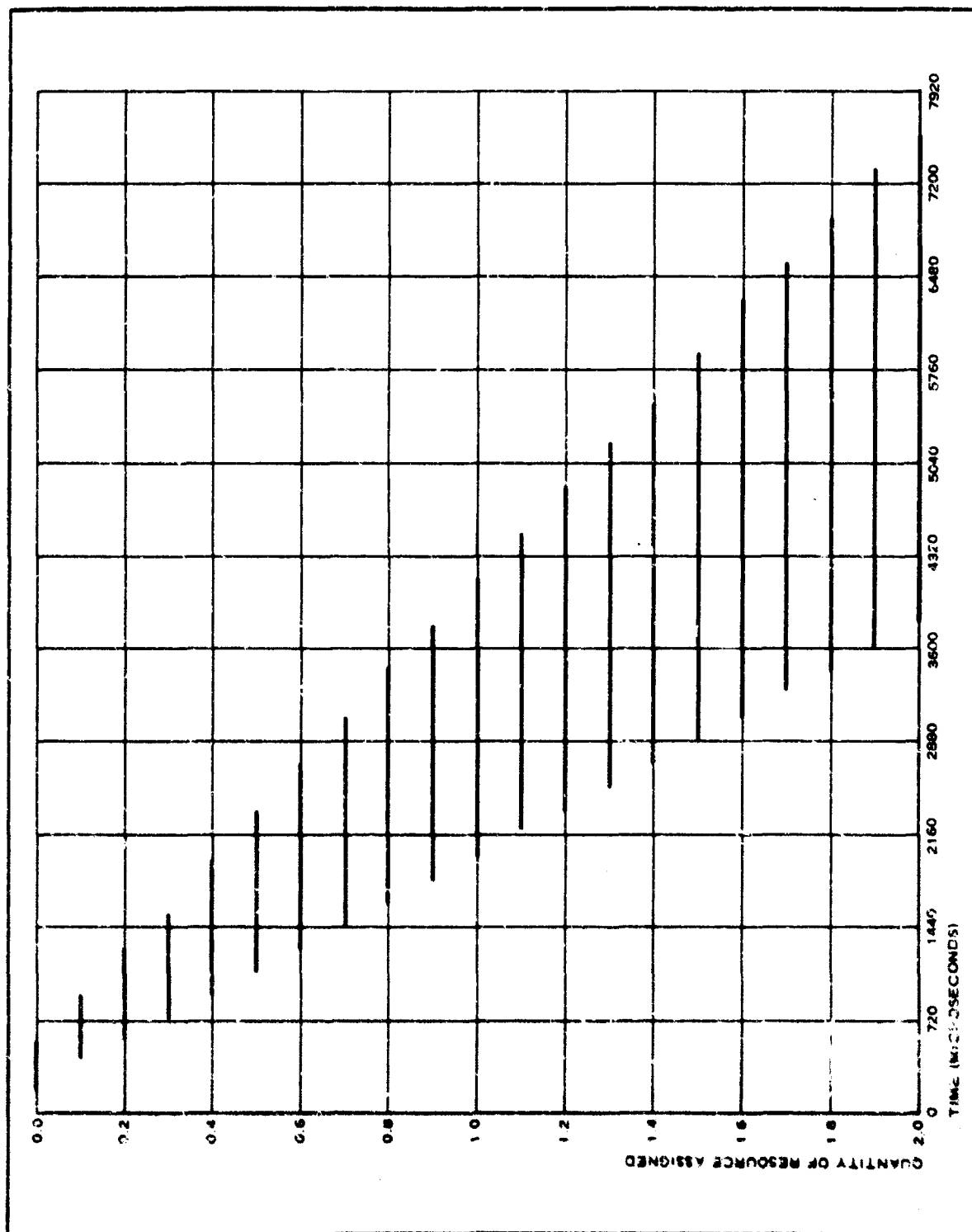
Figure III-16 - Maximization Program 1 Timing Chart, Machine I

Figure III-17 - Maximisation Program 2 Timing Chart. Machine I
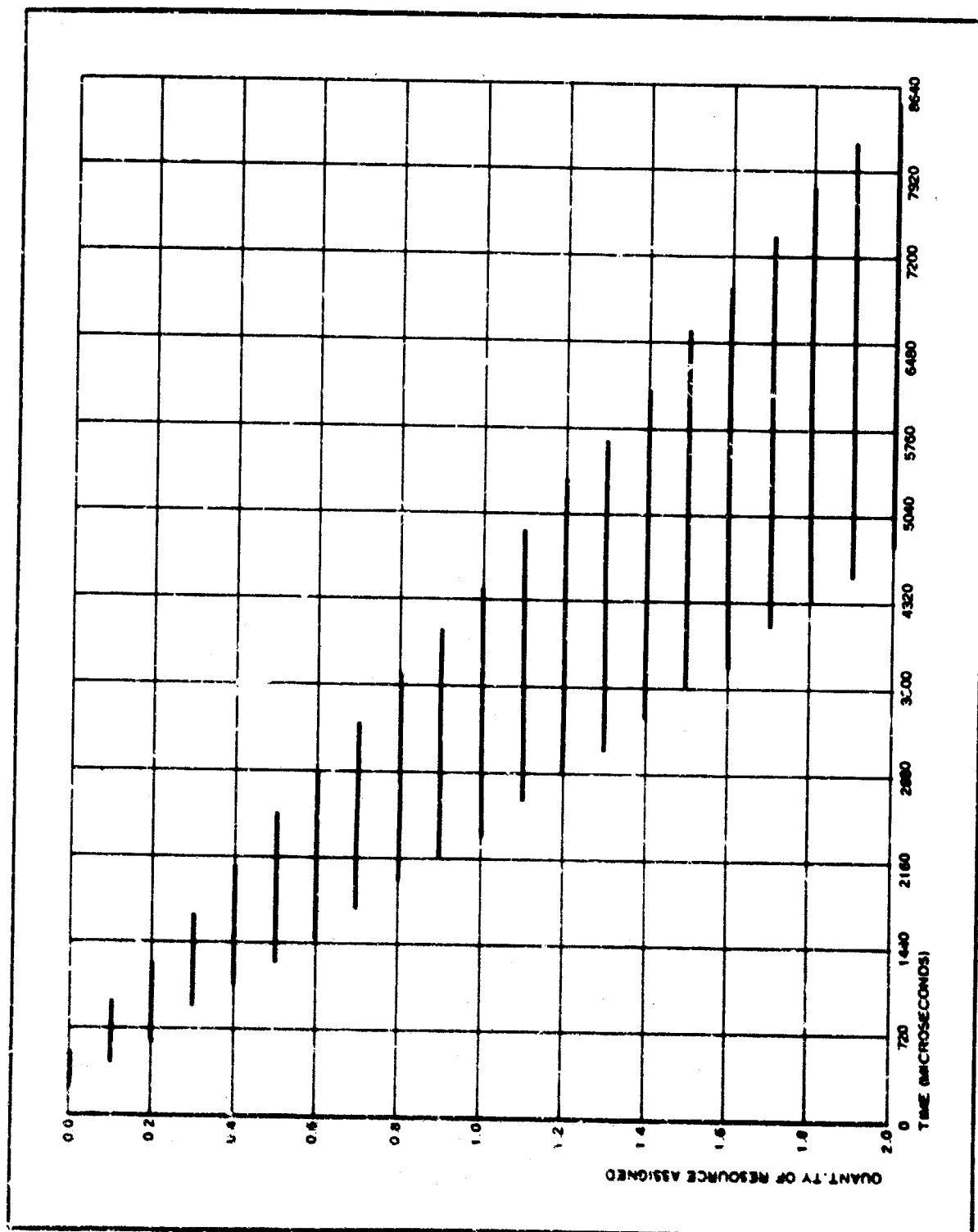
Figure III-18 - Maximization Program 3 Timing Chart, Machine I

problem would be to let $u_4$ generate the index sets for $u_5$. Less processor time would be wasted, although there would be a small amount of time at the beginning of $u_4$ and $u_5$ where data was available but no processor operating on it.

## 4.   COMPARISONS AND CONCLUSIONS

The processors used for Machine I are charted in Figure III-21. Table III-16 compares the IBM 7090 sequential (Appendix II) and the Machine I parallel execution times for the dynamic programming problem.

Although the Machine I execution times for functions $g_1(x)$, $g_2(x)$, $g_5(x)$, and $g_6(x)$ are longer than for the 7090, it should be realized that these routines were not tree'd.

### TABLE III-16 - COMPARISON OF IBM 7090 AND MACHINE I
### EXECUTION TIMES, DYNAMIC PROGRAMMING PROBLEM

| Function | 7090 sequential time ($\mu$sec) | | Machine I parallel time ($\mu$sec) |
| --- | --- | --- | --- |
| | Minimum | Maximum | |
| $g_1(x)$ | 276 | 276 | 3,780 |
| $g_2(x)$ | 366 | 872 | 3,780 |
| $g_3(x)$ | 3,756 | 6,823 | 1,980 |
| $g_4(x)$ | 2,984 | 3,993 | 4,320 |
| $g_5(x)$ | 682 | 1,185 | 4,560 |
| $g_6(x)$ | 335 | 462 | 4,710 |
| Maximization | 140,387 | 208,037 | 16,460 |
| Lookup | 383 | 383 | 510 |
| Total | 0.151 (sec) | 0.224 (sec) | 0.016 (sec) |
| Storage required | 570 | . . . | 734 |

TIME (MICROSECONDS)

QUANTITY OF RESOURCE ASSIGNED

A
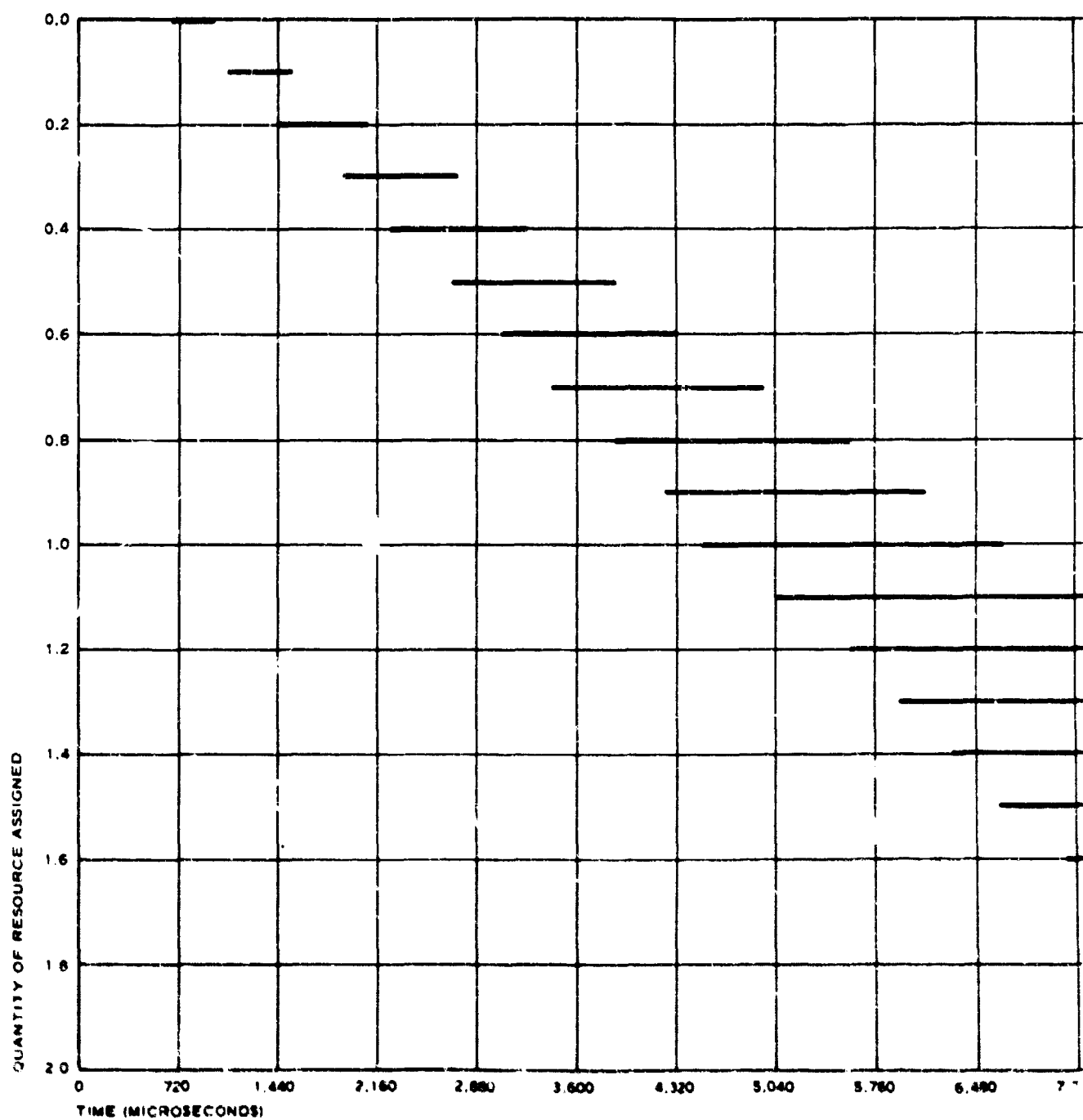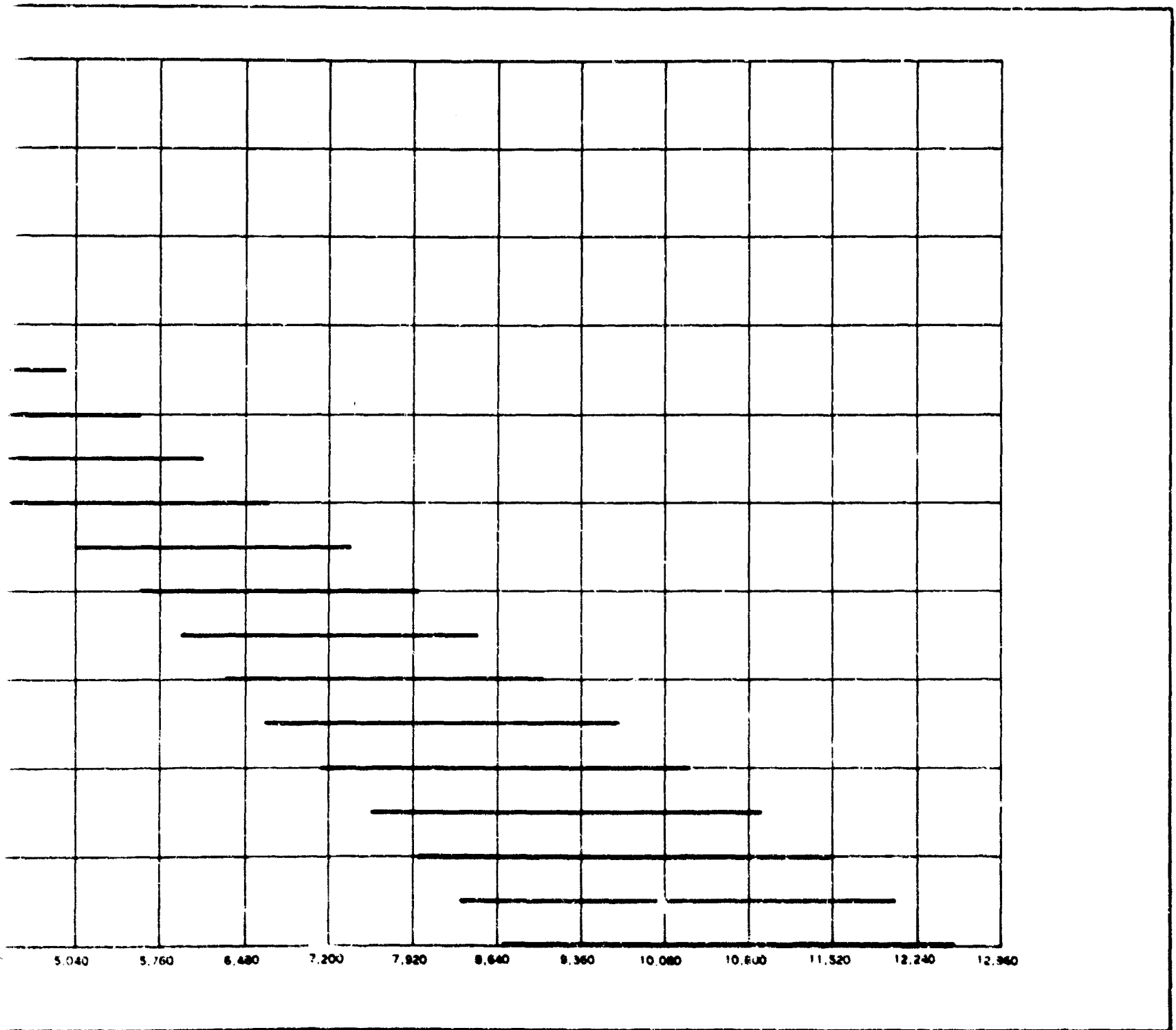
Figure III-19 - Maximization Program 4 Timing Chart. Machine 1

QUANTITY OF RESOURCE ASSIGNED

TIME (MICROSECONDS)
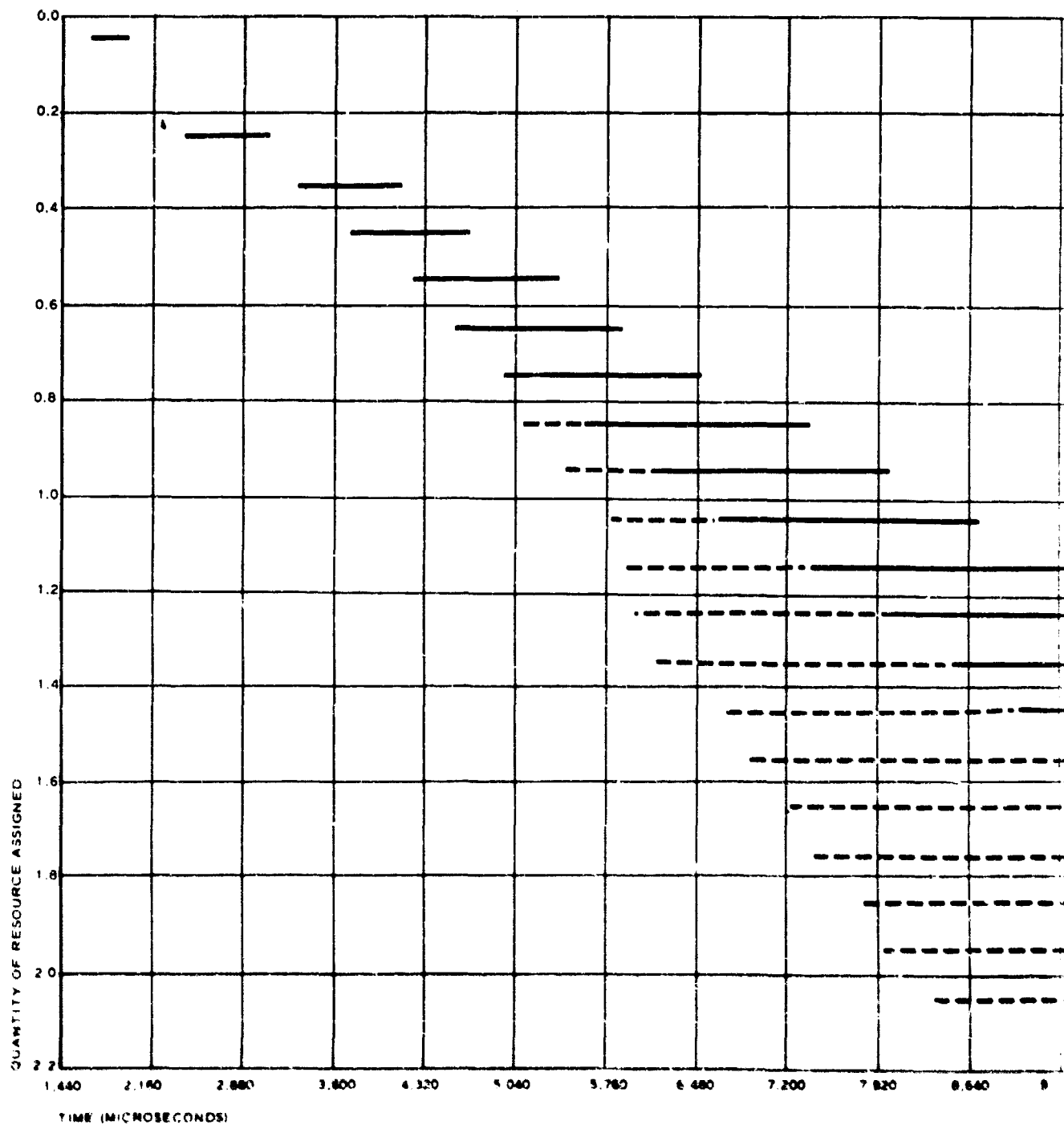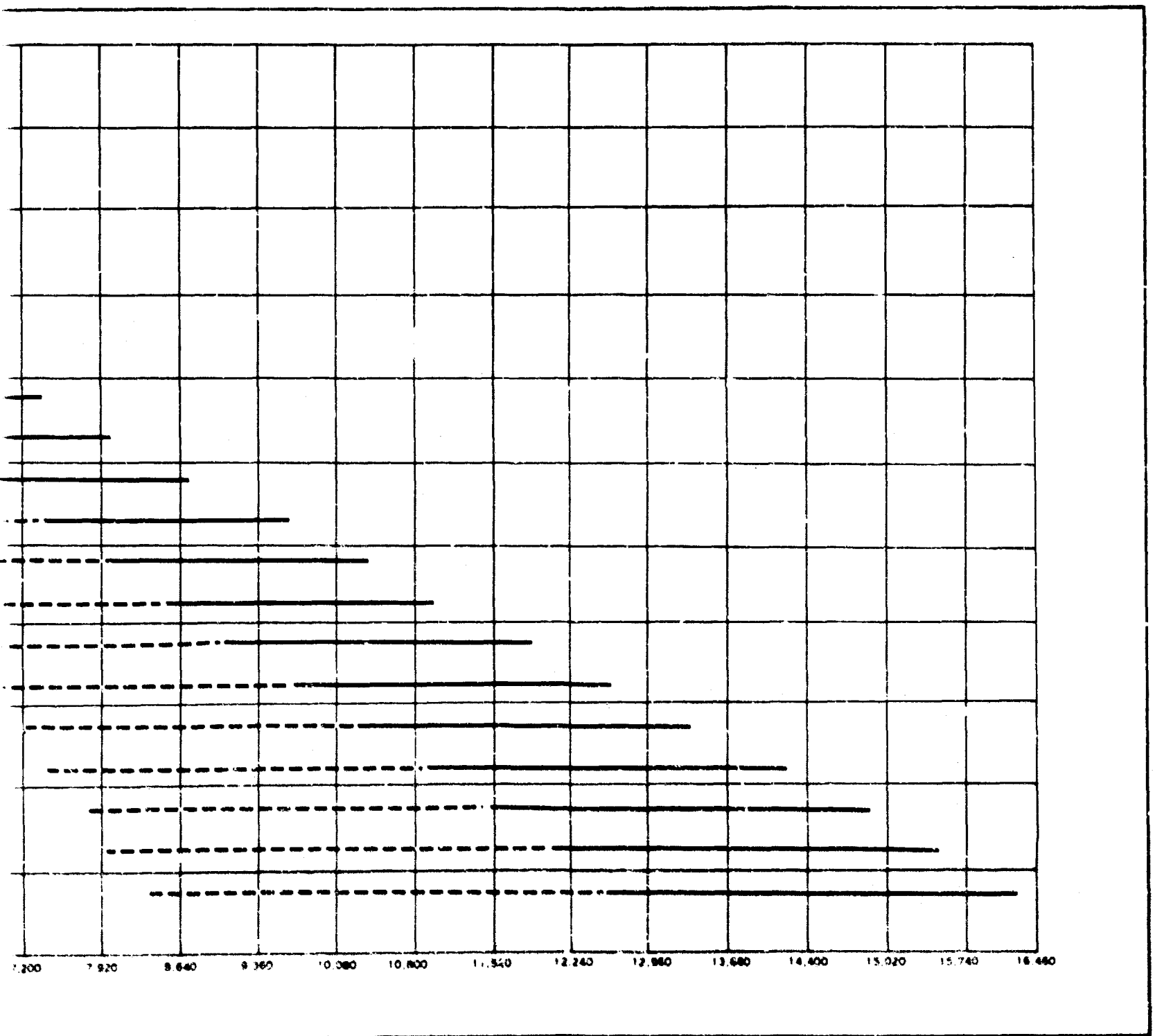
Figure III-20 - Maximization Program 5 Timing Chart, Machine I

B

PROCESSOR USAGE

TIME (MICROSECONDS)

A
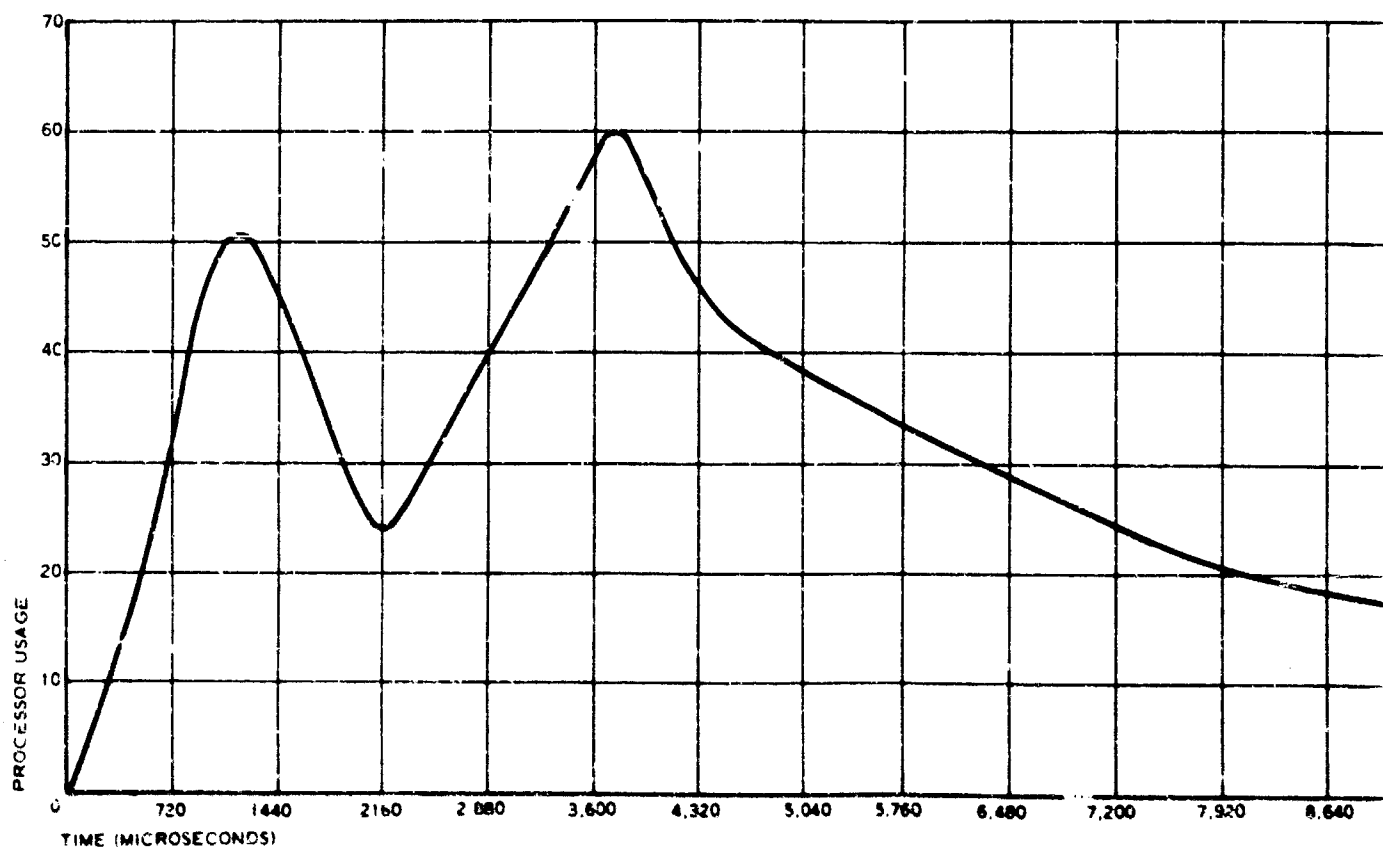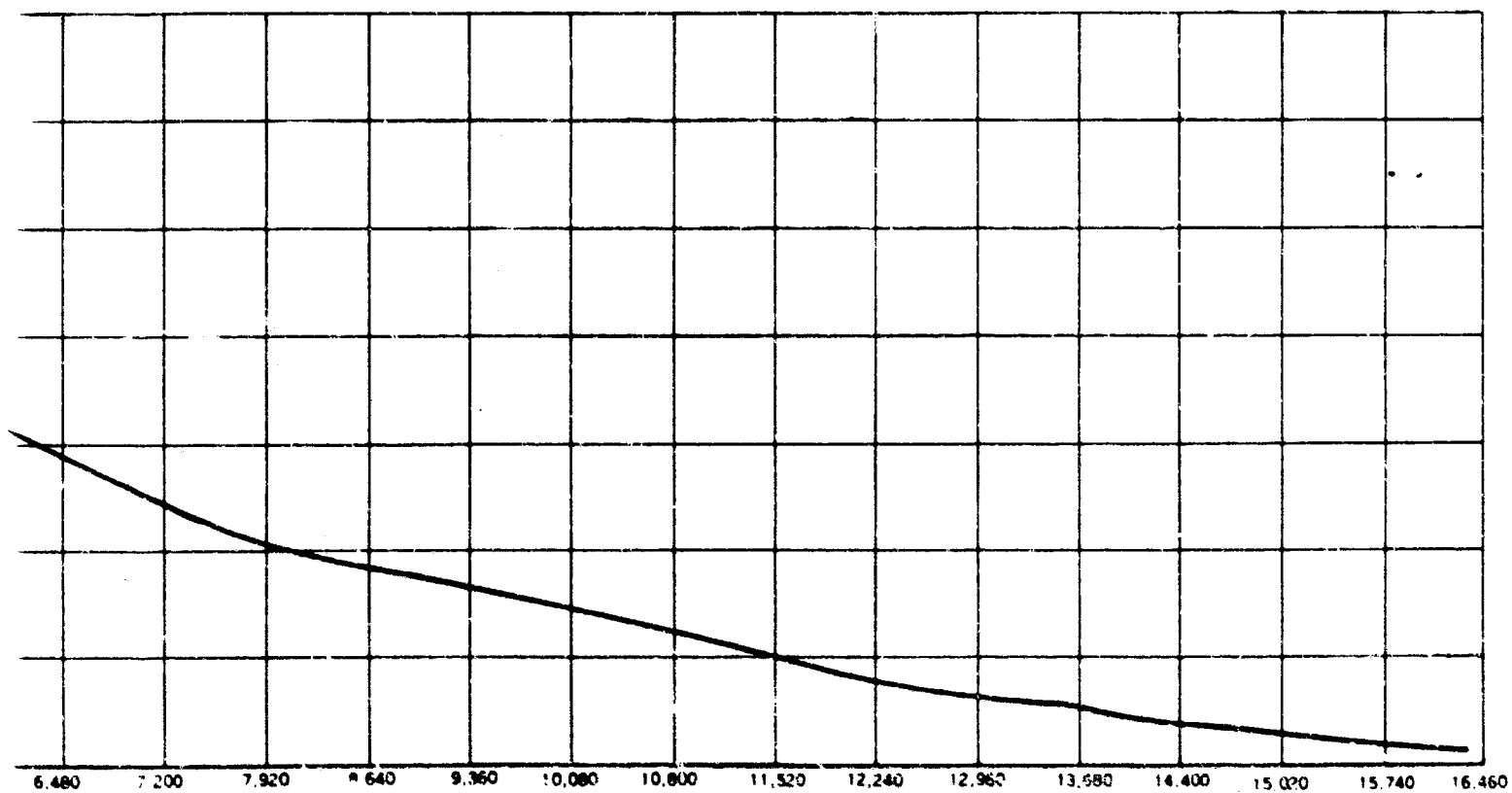
Figure III-21 - Processor-Usage Chart, Machine I

The square root and sine routines were more complex and exhibited parallelism that was extracted. In addition, these two functions were tree'd. The sine routine generated about four times the data that the same routine generated on the sequential machine.

The maximization function takes significantly less time with Machine I than the 7090. The reason is that the Machine I memory is used as the maximizing mechanism. The various combinations of returns are calculated and then sorted by the memory with the largest floating to the top and being retained.

It is relatively easy with Machine I to use a large number of processors on even a small problem. There is a significant amount of parallelism in many problems capable of computer solution. Any iterative sequence where successive loops are independent can be assigned to a group of processors for parallel execution. Independent programs can be executed in parallel and can in turn start any dependent programs as suitable data is generated keeping active processor time to a minimum.

Machine I has an estimated 512 processing units. All processors were assumed to have (1) a program counter, (2) an accumulator, (3) a quotient register, (4) six index registers, and (5) an instruction register.

Each processor has the capability of content addressing any word in memory and can execute interregister transfers. Data can be transferred between processors only via the memory. Machine I fetches and executes two instructions in 30 μsec. The 7090 requires about 8.72 μsec for the same operations. Hence, the 7090 is 3.44 faster than Machine I.

The total time for the solution of the dynamic programming problem on the 7090 ranged from 150 to 220 msec. The total time for problem solution on Machine I was 16 msec. Hence, the Machine I solution time was 9 to 14 times faster than the 7090. Furthermore this particular problem required a maximum of only 60 of the available 512 processors

at any one time. An average of only 22 processors was used. Had the problem been large enough to use the full 512 processors at the peak period, Machine I would then have had a speed advantage of

$$\frac{512}{60} \times (9 - 14) = 76 - 119 \text{ to } 1 .$$

In addition, many processors would have been available at nonpeak times for other uses such as compiling.

## APPENDIX IV - PROGRAMMING MANUAL FOR MACHINE I

1.  MACHINE ORGANIZATION FOR MACHINE I

    The Machine I parallel processor organization used in this study is com-
    posed of 512 logical processing units and 32,768 words of memory. Each
    processing unit has the following registers: program counter, instruction
    register, accumulator, quotient register, and six index registers. The
    program counter is a 24-bit register that is stepped sequentially to gener-
    ate the addresses of the instructions in the program.

    The instruction register is 72 bits long. It is divided into two 36-bit sec-
    tions, upper and lower, each holding a single 36-bit instruction. In most
    cases, an instruction can be located in either the upper or lower half of
    an instruction word, in a few cases, the instruction must be located in
    the upper half of the word.

    The accumulator is a 72-bit register that is considered as one register
    for most instructions; in some cases, it can be treated as two 36-bit
    registers, upper and lower.

    The quotient register also is a 72-bit register; it is treated either as one
    register or two, depending on the instruction being executed.

    Any combination of three of the six index registers included in each proc-
    essing unit may be used. Their contents may be added together with the
    contents of the address field of the instruction to obtain an operand ad-
    dress, an operand, or a shift count.

    If the indices are 1, 2, and 3, there is no reduction in the size of the ad-
    dress field of the instruction. If the indices to be added include 4, 5, or
    6, then the address field is reduced by 6 bits and bit 28 is set to 1. The
    6-bit reduction still leaves an address field of 18 bits. The original 3-bit

index designator field plus two more 3-bit fields are used to contain the 3-bit codes of the specified index registers.

2.  **INSTRUCTIONS**

    **a.**  **Instruction Classes**

    The Machine I parallel processor instructions may be grouped in four classes.

    Class 1 instructions treat the contents of the address field as the operand.   These instructions can be . dexed with any combination of three index registers.   The operand is the result of the addition of the contents of the specified index registers and the contents of the address field of the instruction.

    Class 2 instructions treat the contents of the address field as an operand address.   These instructions can be indexed with any combination of three index registers.   The final operand address is the result of the addition of the contents of the specified index registers and the contents of the address field of the instruction.   The word received by the processor after a fetch type operation is the word next higher than the request word.   When a processor is requesting a word that may or may not be currently available in memory, an NPJ instruction following the fetch will enable a comparison of the names of the requested and received words for an exact match.   If they match, the program continues; if they do not, then a jump is made to the fetch instruction.   A number of desirable operations in this class require execution of two instructions.   This subclass is composed of single- and multiple-erase instructions that require a fetch type instruction following the erase instruction.

    Class 3 instructions treat the contents of the address field as a shift count.   These instructions can be indexed with any combination of three index registers.   The final shift count is the result of the

addition of the contents of the specified index registers and the contents of the address field of the instruction.

Class 4 instructions are special instructions that allow processor unit interregister transfers.

b. Instructions

ENA    $B^n$ y Enter A Lower

The 24-bit operand Y is entered into A lower. The most significant bit of Y is extended in A. Bits 25 to 72 of A are replaced with bit 24 of the operand Y. $Y = y + B^n$. Y is the sum of y and the contents of any combination of three index registers.

ENAU    $B^n$ y Enter A Upper

The 24-bit operand Y $(Y = y + B^n)$ is entered into A upper, bits 37 to 60. The most significant bit of Y is extended in A upper, bits 61 to 72. The contents of A lower are unchanged.

ENQ    $B^n$ y Enter Q Lower

The 24-bit operand Y $(Y = y + B^n)$ is entered into Q lower. Bits 25 to 72 of Q are replaced with bit 24 of the operand Y.

ENBX    $B^n$ y Enter Index Register X

The 24-bit operand Y $(Y = y + B^n)$ is entered into the specified index register, X, with X = 1, 2, 3, 4, 5, 6. This instruction permits the transfer of data from the address field of the instruction to the specified index register or the transfer of the contents of any group of index registers with y added or not to the specified index register.

Example 1:

    (B1) = 2

    (B2) = 3    $B^n = (B2) + (B3)$

    (B3) = -4

    ENB1  B2, B3  6

    The contents of B1 are replaced by 5:

    (B2) + (B3) + y = 3 - 4 + 6 = 5 .

Example 2:

    ENB1  B1, B2, B3  6

    The contents of B1 are replaced by 7:

    (B1) + (B2) + (B3) + y = 2 + 3 - 4 + 6 = 7 .

**INA**    $B^n y$ Increase A Lower

The 24-bit operand Y ($Y = y + B^n$) is added to the least significant 24 bits of A lower. No addition takes place beyond bit 24 of A. The contents of any combination of three index registers can be added to the A register, bits 1 to 24.

**INAU**    $B^n y$ Increase A Upper

The 24-bit operand Y ($Y = y + B^n$) is added to the least significant 24 bits of A upper, bits 37 to 60. No addition takes place beyond bit 60 of A.

**INAE**    $B^n y$ Increase A Lower with Extended Addition

The 24-bit operand Y ($Y = y + B^n$) is added to the contents of the A register. If the contents of the A register lower is an instruction with an address in its address field, an INAE instruction may result in alteration of · instruction operation code and

tag field. Similar changes may occur in A upper if a carry propagates into A upper.

INAUE · $B^n y$ Increase A Upper with Extended Addition

The 24-bit operand Y ($Y = y + B^n$) is added to the contents of A upper. The restrictions on INAE also apply to INAUE.

INB$\underline{X}$ $B^n y$ Increase Index Register X

The 24-bit operand Y ($Y = y + B^n$) is added to the contents of the specified index register, X, with X = 1, 2, 3, 4, 5, 6.

Example 1:

    (B1) = 2

    (B2) = 3

    (B3) = -4

    INB1   B2, B3  6

The contents of B1 are increased by 5:

    $Y = (B2) + (B3) + y = 3 - 4 + 6 = 5$.

Example 2:

    INB1   B1, B2, B3  6

The contents of B1 are increased by 7:

    $Y = (B1) + (B2) + (B3) + 6 = 2 + 3 - 4 + 6 = 7$.

ISKB$\underline{X}$ $B^n y$ Index X Skip

$$X = 1, 2, 3, 4, 5, 6$$

The contents of the specified index register, X, with X = 1, 2, 3, 4, 5, 6, are compared with Y. If the two quantities are equal, the specified index

register is cleared and a full exit is performed. If
the quantities are unequal, the contents of the speci-
fied index register are increased by one, and a half
exit is performed. Normally, this instruction oc-
cupies the upper half of an instruction word. A half
exit then results in execution of the instruction in
the lower half of the instruction word. A full exit
is accomplished by incrementing the program
counter by 1 and executing the upper instruction
of this new instruction word.

**LDA**         $B^n M$ Load A

The contents of A are replaced by the 72-bit operand
contained in storage location M, with $M = (m + (B^n))$.
M, the operand address, is obtained by adding the
contents of the indicated index registers to m.

**LAC**         $B^n m$ Load A Complement

The contents of A are replaced by the complement
of the 72 - bit operand contained in storage location
M, with $M = (m + (B^n))$.

**LDQ**         $B^n m$ Load Q

The contents of Q are replaced by the 72-bit operand
contained in storage location M, with $M = (m + (B^n))$.

**LDBX**         $B^n m$ Load Index Register X

The contents of the specified index register X, with
$X = 1, 2, 3, 4, 5, 6$, are replaced by the least sig-
nificant 24 bits of M, with $M = (m + (B^n))$.

Example:

   (B1) = 2

   (B2) = 3

(B3) = -4

0006

0007    00000000000000000001013

0010

LDB1   B1, B2, B3   6

The contents of B1 are 00001013.

$M = (m + (B1) + (B2) + (B3)) = (6 + 2 + 3 - 4) = (7)$.

ADD        $B^n m$ Add

The 72-bit operand contained in location $M$ is added to the contents of the A register.   $M = (m + (B^n))$.

SUB        $B^n m$ Subtract

The 72-bit operand in location M is subtracted from the contents of A.   $M = (m + (B^n))$.

MLY        $B^n m$ Multiply

The contents of storage location M are multiplied by the contents of the A register.   The 144-bit product is contained in AQ.

DVD        $B^n m$ Divide

The contents of AQ are divided by the contents of storage location M.   The quotient is left in A and the remainder in Q.

FAD        $B^n m$ Floating Add

The sum of two 72-bit floating point operands is formed.   The floating point operand in M is added to the floating point operand in A.   The result is normalized and rounded.

FSB        $B^n$m Floating Subtract

The difference of two 72-bit floating point operands
is formed.  The contents of storage address M are
subtracted from the contents of A.  The result is
rounded and normalized.

FMP        $B^n$m Floating Multiply

The floating point contents of storage location M
are multiplied by the floating points contents of the
A register.  The product is rounded and normalized
in A.

FDV        $B^n$m Floating Divide

The floating point contents of A are divided by the
floating point contents of storage location M.  The
floating point quotient is retained in A.

*STA       $B^n$m Store A

### Note

The asterisk indicates a sig-
nificantly new instruction.

The contents of the A register are re-created in a
memory word with address M.  Every store in-
struction results in the creation of a new word in
memory with an address of M.  It is possible that
a number of words in memory may have the same
address and be ordered according to the contents
of their data fields.  If this situation cannot be tol-
erated then (1) care should be exercised to assign
each word stored in a unique address or (2) all
words in memory with this address should be
erased before the new word is created.  The

ability to store a number of words with identical addresses is desirable since after the next machine cycle the vector of words will be sorted according to their data fields, smallest to the largest.

*STQ $B^n$m Store Q

The contents of the Q register are re-created in a memory word with address M (see +STA instruction above).

*STB$\underline{X}$ $B^n$m Store Index 1

$$X = 1, 2, 3, 4, 5, 6$$

The contents of the specified index register, X, with $X = 1, 2, 3, 4, 5, 6$, are re-created in a memory word with address M. The 24-bit contents of the index register are stored in the least significant 24 bits of the lower portion of the word.

Example:

    (B1) = 2

    (B2) = 3

    (B3) = -4

    STB1   B1, B2. B3   6

The contents of index register 1 are stored in location 7:

0007    000000000000000000000002

If there had been other words in memory with an address of 0007 then the newly created word may have been another element of the vector of words with address 0007:

0007    000000000000000000000000

| | |
|---|---|
| 0007 | 000000000000000000000002 |
| 0007 | 000000000000000000000006 |

**ARS**   $B^n k$ A Right Shift

The contents of A are shifted right K places, with $K = k + (B^n)$. The sign is extended and the lower bits discarded.

**QRS**   $B^n k$ Q Right Shift

The contents of Q are shifted right K places. The sign is extended and the lower bits discarded. $K = k + (B^n)$

**LRS**   $B^n k$ Long Right Shift

The contents of AQ are shifted right K places. The sign of A is extended and the lower-order bits of A replace the higher order bits of Q. The lower order bits of Q are discarded. $K = k + (B^n)$

**ALS**   $B^n k$ A Left Shift

The contents of A are shifted left circular K places. The higher-order bits of A replace the lower order bits. $K = k + (B^n)$

**QLS**   $B^n k$ Q Left Shift

The contents of Q are shifted left circular K places. The higher order bits of Q replace the lower order bits. $K = k + (B^n)$

**LLS**   $B^n k$ Long Left Shift

The contents of AQ are shifted left circular K places. The higher-order bits of A replace the lower-order bits of Q. The higher-order bits of Q replace the lower order bits of A. $K = k + (B^n)$

SCA     $B^n$k Scale A

The contents of A are shifted left circularly until the most significant bit is to the right of the sign bit or until k = 0. Shift count k is reduced by one for each shift and terminates when k = 0, or when the most significant bit is to the right of the sign bit. Upon termination, the count (scale factor) is entered in the specified order register. $K = k + (B^n)$

SCQ     $B^n$k Scale Q

The contents of AQ are shifted left circularly until the most significant bit is to the right of the sign bit of A. Shift count k is reduced by one for each shift. The operation terminates when k = 0 or when the most significant bit is to the right of the sign bit. Upon termination the count (scale factor) is entered in the specified index register. $K = k + (B^n)$

AND     $B^n$m Logical AND

The contents of A are replaced by the logical AND of Q and the contents of M. $M = m + (B^n)$

OR     $B^n$m Logical OR

The contents of A are replaced by the logical OR of Q and the contents of M. $M = m + (B^n)$

EOR     $B^n$m Exclusive OR

The contents of A are replaced by the exclusive OR of the contents of Q and the contents of M. $M = m + (B^n)$

JMP $\quad$ $B^n m$ Jump

Program control is transferred to location M. Normally, the program counter is incremented by one for each instruction word executed. In the case of a jump instruction, the contents of the program counter are replaced by M and the instruction at this address is executed next. $M = m \mp (B^n)$

JNGA $\quad$ $B^n m$ Jump if A Is Negative

Program control is transferred to location M if the sign of A is negative. If the sign of A is positive the next sequential instruction is executed. $M = m + (B^n)$

JNZA $\quad$ $B^n m$ Jump if A Is Nonzero

Program control is transferred to location M if the contents of A are not zero. If the contents of A are zero, the next sequential instruction is executed. $M = m + (B^n)$

JNGQ $\quad$ $B^n m$ Jump if Q Is Negative

Program control is transferred to location M if the contents of Q are negative. If the contents of Q are positive, the next sequential instruction is executed.

JNZQ $\quad$ $B^n m$ Jump if Q Is Nonzero

Program control is transferred to location M if the contents of Q are not zero. If the contents of Q are zero, the next sequential instruction is executed.

COM       $B^n m$ Compare

The contents of A are compared with the contents of M. If (A) are equal to or greater than (M), a half exit is performed. If (A) are less than (M), a full exit is performed. The compare instruction normally is an upper instruction.

BJPX       $B^n m$ Index X Jump

If the contents of the specified index register X, with X = 1, 2, 3, 4, 5, 6, are not zero, the quantity is reduced by one and a jump is executed to location M. If the contents of the specified index register are zero, the next sequential instruction is executed. $M = m + (B^n)$

*NPJ       $B^n m$ Nonpresence Jump

A jump to location M is executed if the address of the word obtained from storage is not the same as the operand address in the instruction. $M = m + (B^n)$

*LDBX

| | | |
|---|---|---|
| UA | Load Index X | Upper A |
| LA | | Lower A |
| UQ | | Upper Q |
| LQ | | Lower Q |

The least significant 24 bits of the upper or lower halves of A or Q are loaded into the specified index register, X, with X = 1, 2, 3, 4, 5, 6.

*ADBX

| | | |
|---|---|---|
| UA | Store Index X | Upper A |
| LA | | Lower A |
| UQ | | Upper Q |
| LQ | | Lower Q |

The contents of the specified index register, X, with $X = 1, 2, 3, 4, 5, 6$, are stored in the least significant 24 bits of the upper or lower half of A or Q.

\*SEH      $\alpha$      Single Erase High

FETCH      $\beta$      Fetch type instruction

The contents of the location which is next larger than the lower limit word $\beta$ are fetched and then erased from memory. $\alpha$ and $\beta$ are the upper and lower limit words that bracket a block of data in memory. The data field of word $\alpha$ is maximum positive while the data field of word $\beta$ is zero. The word fetched and erased is the word next larger than the lower limit word $\beta$.

$$o \gtreqless \beta \quad \alpha = (m + (B^n)!), \ \beta = (m + (B^n))$$

\*SEA

FETCH      Single Erase A

\*SEQ

FETCH      Single Erase Q

The upper limit word $\alpha$ has its contents set to the contents of A or Q. The operation is the same as for SEH.

\*MEH

FETCH      Multiple Erase High

The execution of this instruction is the same as for SEH except that all words between the limits $\alpha$ and $\beta$ are erased.

\*MEA

FETCH      Multiple Erase A

\*MEQ

FETCH      Multiple Erase Q

The execution of these instruction is the same as

for SEA and SEQ except that all words between the limits $\alpha$ and $\beta$ are erased.

*SEHA   Single Erase High A
FETCH

*SEHQ   Single Erase High Q
FETCH

The execution of these instructions is the same as for SEH except that the lower limit word is the same as the contents of A or Q.

*MEH A   Multiple Erase High A
FETCH

*MEAQ   Multiple Erase High Q
FETCH

The execution of these instructions is the same as for MEH except that the. lower limit word is the same as the contents of A or Q.

*SEAA   Single Erase AA
FETCH

*SEQQ   Single Erase QQ
FETCH

The execution of these instructions is the same as for SEA except that the lower limit word is the same as the contents of A or Q.

*SEQA   Single Erase between limits QA
FETCH

*SEAQ   Single Erase between limits ⌐⌐
FETCH

The execution of these instructions is the same as for a SEH except that the upper limit word $\alpha$ is the same as the contents of Q or A and the lower limit word $\beta$ is the same as the contents of A or Q.

*MEAA   Multiple Erase between limits AA
FETCH

| | |
|---|---|
| *MEQQ<br>FETCH | Multiple Erase between Limits QQ |
| | The execution of these instructions is the same as for a SEAA, SEQQ except that all words are erased between the limit words. |
| *MEQA<br>FETCH | Multiple Erase between Limits QA |
| *MEAQ<br>FETCH | Multiple Erase between Limits AQ |
| | The execution of the instructions is the same as for SEQA, SEAQ except that all words are erased between the limit words. |
| *NHA<br>FETCH | Next Higher than A |
| *NHQ<br>FETCH | Next Higher than Q |
| | The word next higher than the lower limit word $\beta$ with contents equal to contents of A or Q is fetched from memory. |
| *BGN $B^n y$ | Begin Y |
| | Y is added to (A) upper. The contents of A are then stored in the WAIT LIST. This instruction is used to start arithmetic units. Prior to the BGN instruction, A has been loaded with an LDI____, JMP PROG instruction. Execution of the BGN results in the addition of Y to the LDI instruction and the storage of this instruction pair in the WAIT LIST. The address that is added is the address of the location where the indices to be transferred are stored. |

## 3. NUMBER REPRESENTATION

A fixed point number consists of a sign bit and coefficient. The upper bit

of a fixed point number designates the sign of the coefficient. If bit 71 is 1, the quantity is negative; a 0 sign bit signifies a positive coefficient. The coefficient may be an integer or fraction. The binary point, in the case of an integer, is assumed to be immediately to the right of the lowest order bit; for a fraction, the point is put to the right of the sign bit.

Floating point numbers are represented by a coefficient and an exponent. The coefficient consists of a 60-bit fraction in the lower 60 positions of the floating point word. The coefficient is a normalized fraction equal to or greater than 1/2 but less than 1. The highest order position, bit 71, is the sign of the coefficient. If the sign bit is 0, the coefficient is positive; if the sign bit is 1, the fraction is negative and in ONE's complement form.

The floating point exponent is an 11-bit quantity with a value ranging from $0000_8$ to $3777_8$. It is formed by adding a true positive exponent and a bias of $2000_8$ or a true negative exponent and a bias of $1777_8$. This results in a range of biased exponents as shown below.

| Three positive exponent | Biased exponent | True negative exponent | Biased exponent |
|---|---|---|---|
| +0 | 2000 | -0 | 2000 |
| +1 | 2001 | -1 | 1776 |
| +2 | 2002 | -2 | 1775 |
| +1776 | 3776 | -1776 | 0001 |
| $+1777_8$ | 3777 | -1777 | 0000 |

4.  BIBLIOGRAPHY

GER-12105:  A Dynamic Programming Program for the IBM 7090.  Akron, Ohio, Goodyear Aerospace Corporation, February 1965.

GER-11777:  A Dynamic Programming Problem.  Akron, Ohio, Goodyear Aerospace Corporation, 12 October 1964.

GER-11949:  Parallel Execution of the Dynamic Programming Technique. Akron, Ohio, Goodyear Aerospace Corporation, 21 January 1965.

## APPENDIX IV

GER-11875: A Parallel Processor Organization. Akron, Ohio, Good-year Aerospace Corporation, 15 December 1964.

# APPENDIX V - BI-TONIC SORTING

## 1. INTRODUCTION

In a previous company report,[a] a new internal sorting method was discussed. Discussed here is another method that, while not as efficient as the referenced method, has certain advantages in parallel processors. To sort $2^n$ words, the method presented here (bi-tonic sorting) requires $n(n + 1)2^{n-2}$ comparisons (with exchanges) while that of the referenced method requires only $(n^2 - n + 4)2^{n-2} - 1$ comparisons.

## 2. BI-TONIC SEQUENCES

Definition: If

$$A = a_1, a_2, \ldots, a_n$$

and

$$B = b_1, b_2, \ldots, b_n$$

are sequences of numbers, then B is a circular permutation of A if and only if there exists an integer, k, with $0 \leq k \leq n - 1$, so that

$$b_i = a_{i + k}$$

for all i's satisfying

$$1 \leq i \leq n - k,$$

and

$$b_i = a_{i + k - n}$$

---

[a]GER-11759. A New Internal Sorting Method. Akron, Ohio, Goodyear Aerospace Corporation, 29 September 1964.

for all i's satisfying

$$n - k + 1 \leqq i \leqq n$$

Definition:

$$A = a_1, a_2, \ldots, a_n$$

is said to be a bi-tonic sequence if there exists a circular permutation of A, $B = b_1, b_2, \ldots, b_n$, and an integer i, with $1 \leqq i \leqq n$, so that

$$b_1 \geqq b_2 \geqq \ldots \geqq b_{i-1} \geqq b_i \leqq b_{i+1} \leqq \ldots \leqq b_{n-1} \leqq b_n .$$

It is easy to see that any monotonic sequence is bi-tonic, as is as the concatenation of any ascending sequence with any descending sequence.

Theorem 1: Any subsequence of a bi-tonic sequence is bi-tonic. Proof: Let A be any bi-tonic sequence and A' any subsequence of A. The theorem need only be proved for any circular permutation of A. Letting $B = b_1, b_2, \ldots, b_n$ be a circular permutation of A where

$$b_1 \geqq b_2 \geqq \ldots \geqq b_{i-1} \geqq b_i \leqq b_{i+1} \leqq \ldots \leqq b_{n-1} \leqq b_n ,$$

it can be seen that any subsequence of B is bi-tonic. This concludes the proof.

Definition: If

$$A = a_1, a_2, \ldots, a_{pq} ,$$

where p and q are integers and $1 \leqq i \leqq p$, then the sequence $A_{i, p}$ is defined

$$A_{i, p} = a_i, a_{i+p}, a_{i+2p}, \ldots, a_{i+(q-1)p} .$$

Definition: If

$$A = a_1, a_2, \ldots, a_{pq} ,$$

then the sequence $A'_{i, p}$ is defined as the sequence $A_{i, p}$ rearranged into ascending order.

Definition: If

$$A = a_1, a_2, \ldots a_{pq},$$

the derived sequence $A^{(j, q)}$ for $1 \leqq j \leqq q$ is defined as a sequence of p terms whose ith term is the jth term of $A'_{i, p}$.

Example: Let $p = 4$, $q = 3$, and $A = 7, 9, 13, 20, 17, 15, 10, 8, 4, 1, 3, 5$. When A is written as terms of a 3- by 4 matrix (across the first row, then across the second row, etc.):

$$\begin{pmatrix} 7 & 9 & 13 & 20 \\ 17 & 15 & 10 & 8 \\ 4 & 1 & 3 & 5 \end{pmatrix},$$

then the first column is the sequence $A_{1, 4} = 7, 17, 4$, the second column is the sequence $A_{2, 4} = 9, 15, 1$, etc. If each column of this matrix is rearranged into ascending order:

$$\begin{pmatrix} 4 & 1 & 3 & 5 \\ 7 & 9 & 10 & 8 \\ 17 & 15 & 13 & 20 \end{pmatrix},$$

then $A'_{1, 4} = 4, 7, 17$ is the first column, $A'_{2, 4} = 1, 9, 15$ is the second column, etc., and $A^{(1, 3)} = 4, 1, 3, 5$ is the first row, $A^{(2, 3)} = 7, 9, 10, 8$ is the second row, etc.

In the above example, A is bi-tonic since one circular permutation of A is 20, 17, 15, 10, 8, 4, 1, 3, 5, 7, 9, 13. As predicted by theorem 1, the subsequences $A_{1, 4} = 7, 17, 4$; $A_{2, 4} = 9, 15, 1$; $A_{3, 4} = 13, 10, 3$, and $A_{4, 4} = 20, 8, 5$, also are bi-tonic. In the example, the derived sequences $A^{(1, 3)} = 4, 1, 3, 5$, $A^{(2, 3)} = 7, 9, 10, 8$, and $A^{(3, 3)} = 17, 15, 13, 20$, are bi-tonic and, furthermore, $A^{(1, 3)}$ has the least four members of A.

$A^{(2,\ 3)}$ has the middle four, and $A^{(3,\ 3)}$ has the greatest four members of A. Hence, to reorganize A into ascending order, it is sufficient to re-arrange each of the three bi-tonic-derived sequences into ascending order and concatenate them. Theorem 2 shows this is true for any bi-tonic sequence.

Theorem 2: If

$$A = a_1,\ a_2,\ \ldots,\ a_{pq}$$

is bi-tonic, then each derived sequence $A^{(j,\ q)}$, where $1 \leqq j \leqq q$, is bi-tonic and

$$\max\left[A^{(1,\ q)}\right] \leqq \min\left[A^{(2,\ q)}\right],$$

$$\max\left[A^{(2,\ q)}\right] \leqq \min\left[A^{(3,\ q)}\right],$$

$$\cdot$$
$$\cdot$$
$$\cdot$$

$$\max\left[A^{(q-1,\ q)}\right] \leqq \min\left[A^{(q,\ q)}\right].$$

Proof: Consider A written in matrix form:

$$\begin{pmatrix} a_1 & a_2 & a_3 & \cdots & a_p \\ a_{p+1} & a_{p+2} & a_{p+3} & \cdots & a_{2p} \\ a_{2p+1} & a_{2p+2} & a_{2p+3} & \cdots & a_{3p} \\ \cdot & & & & \\ \cdot & & & & \\ \cdot & & & & \\ a_{(q-1)p+1} & a_{(q-1)p+2} & a_{(q-1)p+3} & \cdots & a_{pq} \end{pmatrix}$$

and observe that a circular permutation of A is equivalent to a circular permutation of the columns plus a circular permutation within each column. The effect of a circular permutation within any column is cancelled when the column is rearranged into ascending order. Hence, a circular permutation of A causes a circular permutation within each derived sequence; this does not affect the bi-tonic property and maximums and minimums. It is concluded that it is sufficient to prove the theorem for any circular permutation of A.

Pick a circular permutation of A, $B = b_1, b_2, \ldots, b_{pq}$ for which there is an integer j, with $1 \leq j \leq pq$, so that

$$b_1 \geq b_2 \geq \ldots \geq b_{j-1} \geq b_j \leq b_{j+1} \leq \ldots \leq b_{pq-1} \leq b_{pq} .$$

Let r and s be the integers defined by

$$rp + s = j \text{ and } 1 \leq s \leq p .$$

B in matrix form is

$$
\begin{pmatrix}
b_1 & b_2 & \cdots & b_s & \cdots & b_p \\
b_{p+1} & b_{p+2} & \cdots & b_{p+s} & \cdots & b_{2p} \\
\bullet & & & & & \\
\bullet & & & & & \\
\bullet & & & & & \\
b_{rp+1} & b_{rp+2} & \cdots & b_{rp+s} & \cdots & b_{(r+1)p} \\
\bullet & & & & & \\
\bullet & & & & & \\
\bullet & & & & & \\
b_{(q-1)p+1} & b_{(q-1)p+2} & \cdots & b_{(q-1)p+s} & \cdots & b_{pq}
\end{pmatrix}
$$

It is easy to see that for any k, with $1 \leq k \leq p$, $\max \left[ b_k, b_{p+k}, b_{2p+k}, \ldots, b_{(q-1)p+k} \right] = \max \left[ b_k, b_{(q-1)p+k} \right]$, so that after each column is rearranged into ascending order each term of row q the derived sequence $B^{(q, q)}$ comes from row 1 or row q. The proof is divided into three cases; $0 < r < q-1$, $r = 0$, and $r = q - 1$.

For $0 < r < q - 1$, the inequalities

$$b_1 \geq b_2 \geq b_3 \geq \ldots \geq b_p$$

and

$$b_{(q-1)p+1} \leq b_{(q-1)p+2} \leq \ldots \leq b_{pq}$$

hold, so if $b_k \leq b_{(q-1)p+k}$ for some k, where $1 = k \leq p$, then $b_{k+1} \leq b_{(q-1)p+k+1}$. This, together with $B^{(q, q)} = \max \left[ b_1, b_{(q-1)p+1} \right]$, $\max \left[ b_2, b_{(q-1)p+2} \right], \ldots, \max(b_p, b_{pq})$ implies that for some integer t, where $0 \leq t \leq p$, $B^{(q, q)} = b_1, b_2, \ldots, b_{t-1}, b_t, b_{(q-1)p+t+1}, b_{(q-1)p+t+2}, \ldots, b_{pq}$.

For $1 \leq t \leq p - 1$, $B^{(q, q)}$ is bi-tonic and its minimum is

$$\min \left[ b_t, b_{(q-1)p+t+1} \right].$$

Let $C = b_{t+1}, b_{t+2}, \ldots, b_{(q-1)p+t}$; then C is bi-tonic with a maximum of

$$\max \left[ b_{t+1}, b_{(q-1)p+t} \right]$$

and

$$b_t \geq b_{t+1},$$

$$b_t \geq b_{(q-1)p+t},$$

$$b_{(q-1)p+t+1} \geq b_{t+1},$$

and

$$b_{(q - 1)p + t + 1} \geqq b_{(q - 1)p + t}$$

are established so $\min \left[B^{(q, \; q)}\right] \geqq \max (C)$. If $t = 0$ or $t = p$, $B^{(q, \; q)}$ is monotonic, hence bi-tonic, C is bi-tonic, and $\min \left[B^{(q, \; q)}\right] \geqq \max (C)$.

The $r = 0$ case can be reduced to the case $r = q - 1$ by inverting the order of the terms of B; this operation inverts each derived sequence and does not affect the bi-tonic property, maximums and minimums.

For $r = q - 1$,

$$b_1 \geqq b_2 \geqq \ldots \geqq b_p \geqq \ldots \geqq b_{(q - 1)p + 1} \geqq b_{(q - 1)p + 2} \geqq \ldots$$

$$\geqq b_{(q - 1)p + s - 1} \geqq b_{(q - 1)p + s} \leqq b_{(q - 1)p + s + 1} \leqq \ldots$$

$$\leqq b_{pq - 1} \leqq b_{pq} .$$

If $b_{pq} \leqq b_p$, then $B^{(q, \; q)} = b_1, \; b_2, \; \ldots , b_p$, which is monotonic, hence bi-tonic, with a minimum of $b_p$. $C = b_{p + 1}, \; b_{p + 2}, \; \ldots , \; b_{pq}$ is bi-tonic with a maximum of $\max (b_{p + 1}, \; b_{pq})$, so $\min \left[B^{(q, \; q)}\right] \geqq \max (C)$.

If $b_{pq} > b_p$, then there is an integer t, $s \leqq t \leqq p - 1$, so that $b_t \geqq b_{(q - 1) p + t}$ and $b_{t + 1} \leqq b_{(q - 1)p + t + 1}$. $B^{(q, \; q)} = b_1, \; b_2, \; \ldots , \; b_t,$ $b_{(q - 1)p + t + 1}, \; b_{(q - 1)p + t + 2}, \; \ldots , \; b_{pq}$ is bi-tonic with a minimum of

$$\min \left[b_t, \; b_{(q - 1) p + t + 1} \right] .$$

$C = b_{t + 1}, \; b_{t + 2}, \; \ldots , \; b_{(q - 1)p + t}$ is bi-tonic with a maximum of $\max \left[b_{t + 1}, \; b_{(q - 1)1 + t} \right]$ and again $\min \left[B^{(q, \; q)}\right] \geqq \max (C)$. This concludes the case $r = q - 1$.

In all three cases, $B^{(q, \; q)}$ is bi-tonic and if $C = B - B^{(q, \; q)}$, C is bi-tonic and $\min \left[B^{(q, \; q)}\right] \geqq \max (C)$. C has $(q - 1)p$ terms. It is not hard to see that the derived sequences $C^{(1, \; q - 1)}, \; C^{(2, \; q - 1)}, \; \ldots , C^{(q - 1, q - 1)}$ of C will be circular permutations of $B^{(1, \; q)}, \; B^{(2, \; q)}, \; \ldots , \; B^{(q - 1, \; q)}$, respectively. Hence, the above proof could be carried out on C to show

that $C^{(q-1,\ q-1)}$ $\left[\text{and therefore } B^{(q-1,\ q)}\right]$ is bi-tonic and min $\left[C^{(q-1,\ q-1)}\right] \geq \max\left[C - C^{(q-1,\ q-1)}\right]$. Therefore, min $\left[B^{(q-1,\ q)}\right] \geq \max\left[B - B^{(q,\ q)} - B^{(q-1,\ q)}\right]$. Iteration of this process proves theorem 2.

## 3. BI-TONIC SORTING OPERATORS

For any integer $n > 1$, let $N_n$ be an operator that, when applied to any bi-tonic sequence of length n, causes the terms of it to be rearranged into ascending order. Theorems 1 and 2 show that for any integers $p > 1$ and $q > 1$, $N_{pq}$ can be constructed from p applications of $N_q$ and q applications of $N_p$. The operator equation is

$$N_{pq}(A) = N_q(A_{1,\ p})N_q(A_{2,\ p}) \ldots N_q(A_{p,\ p})N_p\left[A^{(1,\ q)}\right]N_p\left[A^{(2,q)}\right] \ldots N_p\left[A^{(q,\ q)}\right].$$

Or, using $\prod$ notation,

$$N_{pq}(A) = \prod_{i=1}^{p} N_q(A_{i,\ p}) \cdot \prod_{j=1}^{q} N_p\left[A^{(j,\ q)}\right].$$

A special case is when $p = q^{t-1}$,

$$N_{q^t}(A) = \prod_{i=1}^{q^{t-1}} N_q(A_{i,\ q^{t-1}}) \cdot \prod_{j=1}^{q} N_{q^{t-1}}\left[A^{(j,\ q)}\right].$$

Repeated application of this equation allows construction of $N_{q^t}$ from $N_q$ operators. When $q = 2$,

$$N_{2^t}(A) = \prod_{i=1}^{2^{t-1}} N_2(A_{i,\ 2^{t-1}}) \cdot N_{2^{t-1}}\left[A^{(1,\ 2)}\right] \cdot N_{2^{t-1}}\left[A^{(2,\ 2)}\right].$$

The $N_2$ operator is a comparison of two numbers with an exchange if they are in the wrong order ($N_2$ is the same as $Q_2$ in GER-11759).[a] The $N_{2^t}$ operator will have $t2^{t-1}$ $N_2$ operators.

---

[a]Ibid.

A bi-tonic sequence can be formed from any two ascending sequences by inverting one sequence and concatenating them. The bi-tonic sequence then can be sorted by means of an N operator; the result is the merge of the ascending sequences. Hence, the operator $N_n$ is equivalent to $M_{m, k}$ of the referenced report for any m, k where $m + k = n$.

In general, $N_{m + k}$ uses more $Q_2$ operators than $M_{m, k}$. This is the price to obtain a merge operator that is dependent only on $m + k$ and not on m and k separately; for example, in the referenced report, $M_{2^t - 1, 2^t - 1}$ uses $(t - 1)2^{t - 1} + 1$ operators but $N_{2^t}$ has the advantage that it can be used to merge any two ascending sequences whose combined length is $2^t$ where $M_{2^t - 1, 2^t - 1}$ can only be used when both sequences have the same length, $2^t - 1$.

If a sort of $2^n$ numbers is conducted using N operators for merging, the sort will use $(n^2 + n)2^{n - 2}$ $Q_2$ operators whereas, in the referenced report, only $(n^2 - n + 4)2^{n - 2} - 1$ $Q_2$ operators are required using M operators.

## 4. CONCLUSIONS

This report indicates how a merge operator $N_{2^t}$ can be constructed from $Q_2$ operators that will merge any two ascending sequences whose combined length is $2^t$. $N_{2^t}$ is more versatile than $M_{m, 2^t - m}$ of the referenced report because of this but it uses slightly more $Q_2$ operators.

# APPENDIX VI - BASIC ORGANIZATION OF MACHINE I

1. ## INTRODUCTION

   This report describes a computer organization consisting of several
   arithmetic units and several I/O channels interconnected by a multiaccess
   self-sorting memory. All arithmetic units and I/O channels can access
   the memory at the same time with no conflict, even when two or more
   units access the same word. The sorting capability of the memory allows
   fast sorting and searching of tables and a form of content addressing.
   This capability, together with the parallel arithmetic capability, gives the
   processor a fast processing speed on most classes of problems.

2. ## THE PROBLEM OF ACCESSING DATA IN COMPUTER ORGANIZATIONS

   Initially, higher processing speeds in digital computers were obtained
   mostly by using faster components. Later, higher speeds were obtained
   mostly by doing operations simultaneously that previously were done one
   at a time; for example, the first computers suspended computations during
   I/O operations while later machines do both simultaneously. The latest
   large-scale processors, such as the IBM Stretch and the CDC-6600, allow
   several I/O and arithmetic operations to take place simultaneously.

   In the future, computers with a large number of simultaneously operating
   arithmetic units (ALU's) and I/O channels, perhaps in the hundreds, can
   be expected. The major problem in such a computer is that of giving all
   units fast access to the data that they need.

   This problem exists in present-day computers where memories are di-
   vided into several banks so that while one channel is accessing one memory
   bank other channels can be accessing other banks. When two or more
   channels need access to the same memory bank, one of the channels is

-139-

permitted access and the others wait. It can be expected that the fre-
quency of these conflicts will be very high if there are hundreds of chan-
nels, and thus this method does not appear to be promising. This is
especially true if data are retrieved by content rather than by address,
since a given item might be in any memory bank and a channel must look
in all the memory banks for the item. In this case, dividing the memory
into banks does little to increase the overall processing speed.

When the class of problems to be solved by a particular machine is re-
stricted, the machine can be tailored to prevent memory conflicts. An
example of this is the SOLOMON computer[1, a] where each processing
element is allowed access only to its four neighbors (right, left, up, down).
On certain problems with a rectangular structure (matrices, partial-
difference equations, etc.), the SOLOMON computer achieves a fast
processing speed because each processing element only needs access to
its four neighbors while for other problems the time spent in shuffling
operands to the elements needing them will slow the processing speed
drastically.

The above discussion exhibits the need for a memory capable of perform-
ing hundreds of accesses simultaneously without conflicts. Ideally, no
conflicts should arise even when several channels want the same word.
This situation exists if several ALU's jump simultaneously to a common
subroutine such as a square root subroutine.

3.  A MULTIACCESS SELF-SORTING MEMORY ORGANIZATION

a.  Introduction

One way to build a memory with m words and n access lines is to use a
matrix or crossbar switch with m rows and n columns. The amount of
equipment in such an arrangement is proportional to mn, a prohibitive

---

[a] Superior numbers in the text refer to references listed under Subhead 6 on
page 162.

number for memories of reasonable size. Another disadvantage of the matrix is the fan-out and fan-in required of some of its elements; for example, there is a gate for each word in memory loading down each access line. The fan-out and fan-in can be reduced by "treeing" but this increases the amount of equipment even more.

Fortunately, other networks of elements exist that perform the same function as the m-by-n crossbar; these use only $m/2 \log_2 n$ elements (approximately) and the fan-in and fan-out required of each element is constant regardless of m and n. These networks are based on the sorting and merging techniques discussed in GER-11759[2] and GER-11869[3] and described below.

b. The Comparison Element

Each element in the sorting and merging networks has two inputs, A and B, and two outputs, L and H, as shown in Figure VI-1. When two items of data are presented on the inputs, the element compares the two items as if they are numbers and presents the lower of the two on output L, and the higher on output H. If the two items are equal, the element presents their common value on both outputs. Either bit-serial or bit-parallel or a serial-parallel form of data transmission is possible; however, serial transmission should be done most-significant bit first.

Figure VI-2 shows a 13-NOR comparison element through which data are transmitted one bit at a time, most-significant bit first. Basically, this element operates as follows. The B > A and A > B flip-flops are reset by the reset input and then the data items are presented on A and B serially, most-significant bit first, interspersed with clock pulses on the clock input. With the flip-flops reset, the L output is the logical product (AND) of A and B, while H is the logical union (OR). If A = B, the clock pulse has no effect. If A = 1 and B = 0, the A > B flip-flop is set. If A = 0 and B = 1, the B > A flip-flop is set. If the B > A flip-flop is set, it remains so until the next reset pulse. It changes the operation of the circuit

IF A $\geq$ B, THEN L = B AND H = A

IF A $\leq$ B, THEN L = A AND H = B
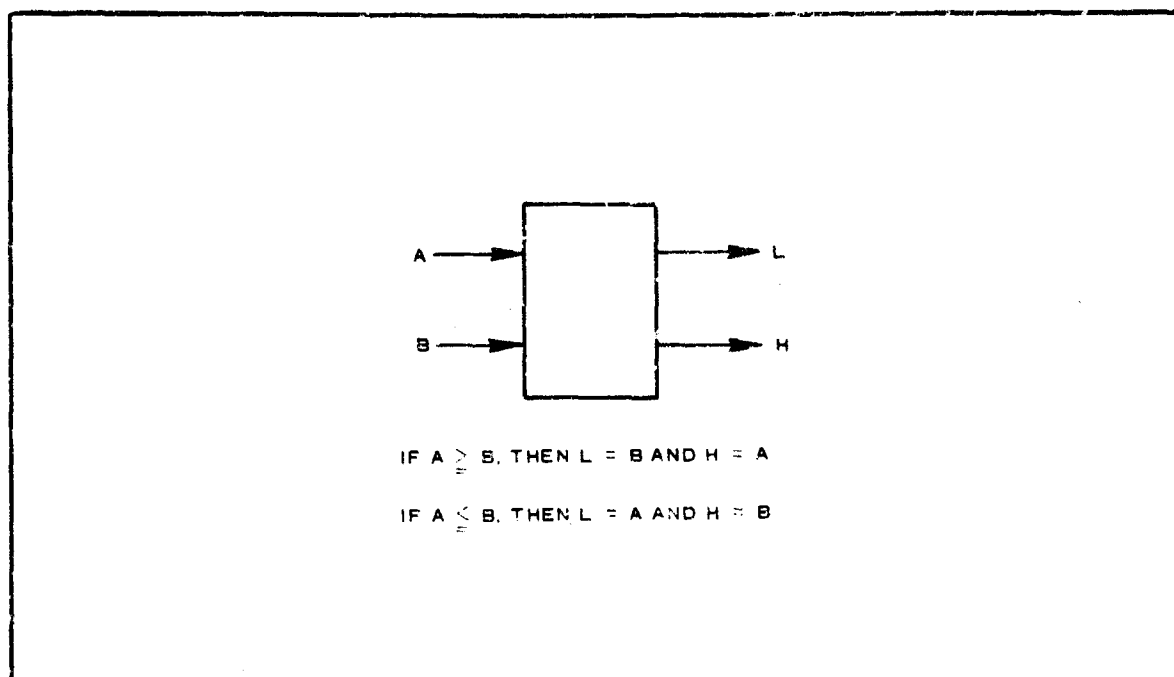
Figure IV-1  Symbol for a Comparison Element

so that L = A and H = B and it inhibits the setting of the other flip-flop. The operation is similar if the A > B flip-flop is set.

Comparison elements that compare more than one bit of each item at a time also are possible. Also, it is possible to add a shift-register stage to each input or to each output so that the element has a temporary storage function as well as a comparison function.

In the networks to be described, the L and H output of each element will be connected to an A or B input of another element, hence the load on the L and H outputs is fixed. This should make it possible to construct these elements economically; for instance, the logic of Figure VI-2 could be put on one integrated circuit chip so that the elements could be fabricated in batches.

c.  $M_{m,n}$ Merging Networks

The comparison elements can be combined to form a network that can merge an ordered set of m items with an ordered set of n items to form
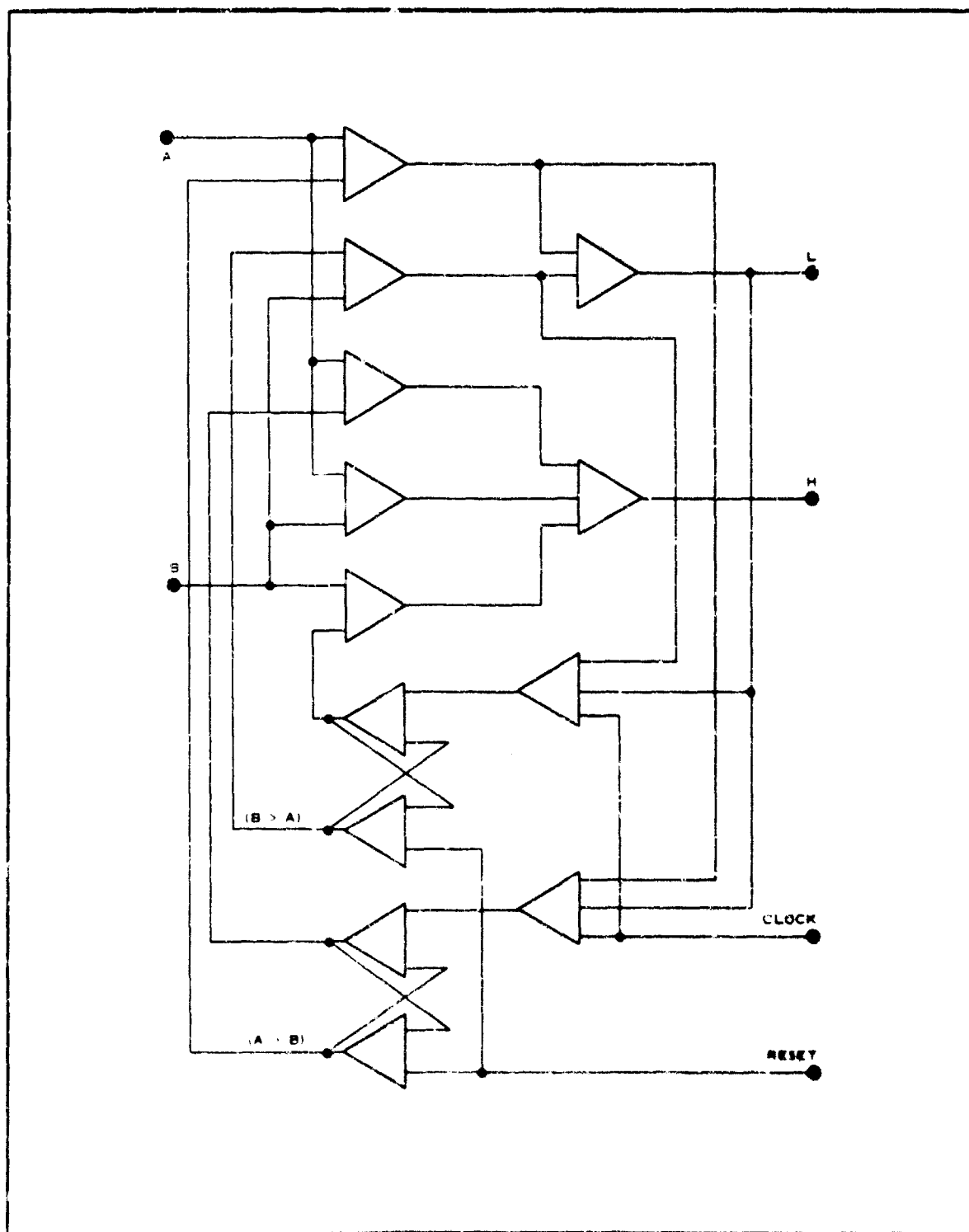
Figure VI-2 - A 13-NOR Comparison Element

an ordered set of $m + n$ items (Figure VI-3); that is, if $m$ items arrive over input lines $a_1$, $a_2$, . . . , $a_m$ and $n$ items arrive over $b_1$, $b_2$, . . . . , $b_n$ simultaneously and if $a_1 \leqq a_2 \leqq$ . . . $\leqq a_m$ and $b_1 \leqq b_2 \leqq$ . . . $\leqq b_n$, then the $m + n$ items will be sent out on $c_1$, $c_2$, . . . , $c_{m+n}$ reordered so that $c_1 \leqq c_2 \leqq$ . . . $\leqq c_{m+n}$. This is called an $M_{m,\,n}$ merging network.

The construction of an $M_{m,\,n}$ merging network is based on the merging technique described in Reference 2. Basically, the network merges the set $a_1$, $a_3$, $a_5$, . . . with the set $b_1$, $b_3$, $b_5$, . . . in one subnetwork while merging the set $a_2$, $a_4$, $a_6$, . . . with the set $b_2$, $b_4$, $b_6$, . . . in another subnetwork. The outputs of the two subnetworks are combined to form the output $c_1$, $c_2$, $c_3$, . . . . The subnetworks in turn each consist of two subnetworks combined the same way, etc.

The construction is made more explicit in Figure VI-4, which shows how two subnetworks are combined to form the larger merging network, $M_{m,\,n}$.
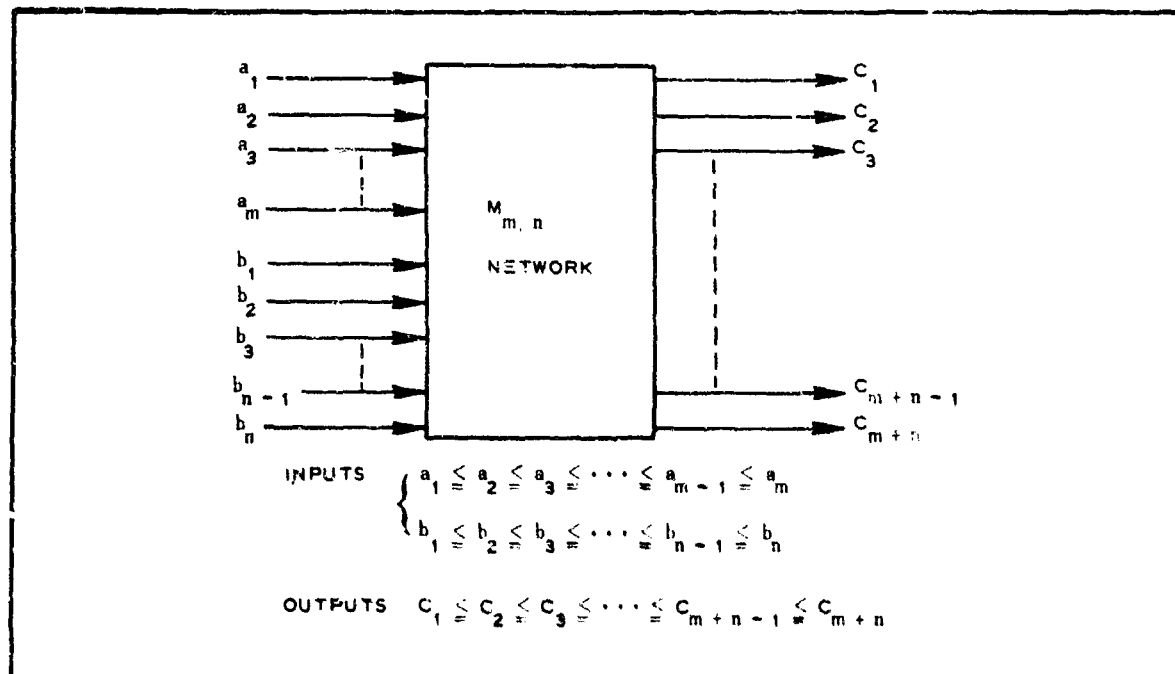


Figure VI-3 - Symbol for an $M_{m,\,n}$ Merging Network
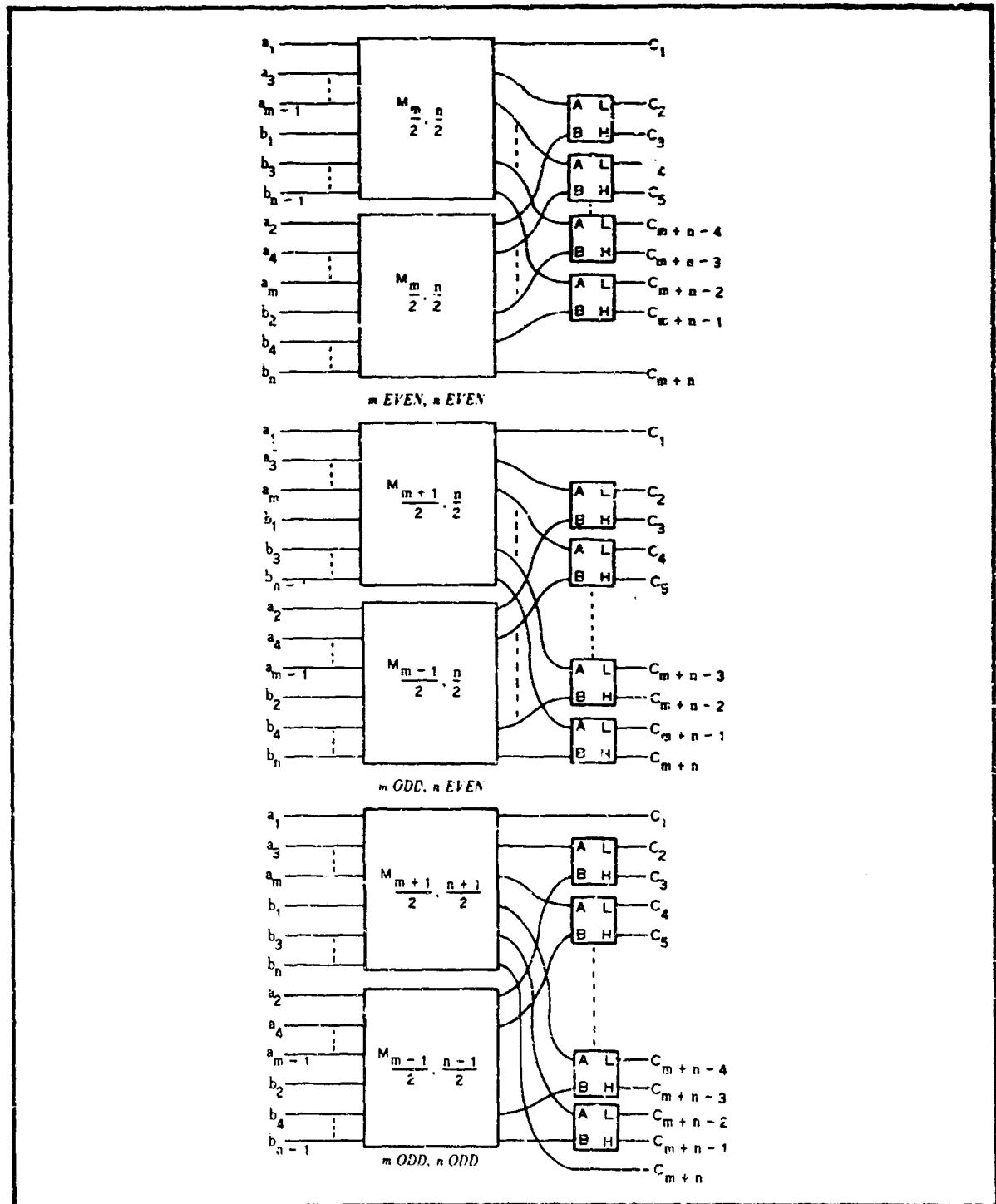
Figure VI-4 - Construction of $M_{m, n}$ from Two Subnetworks and a Set of Comparison Elements

There are four cases depending on the odd-even character of the numbers m and n. The case where m is even and n is odd is obtained from the case where m is odd and n is even simply by interchanging the two input sets; hence, only three cases are illustrated. In all three cases, one subnetwork receives all the even-indexed items of the input sets while the other subnetwork receives the odd-indexed items. The respective outputs of the two subnetworks are compared by a set of comparison elements (one or two of the subnetwork outputs bypass this stage as indicated in Figure VI-4) and the outputs of these elements are the outputs of $M_{m, n}$.

By applying the same procedure to the subnetworks and then to the sub-subnetworks, etc., the construction is reduced to a set of $M_{p, 1}$, $M_{1, p}$ merging networks. An $M_{p, 1}$ network can be built by means of the familiar binary search technique; that is, the item $b_i$ that is to be merged with $a_1$, $a_2$, . . . . $a_p$ is compared first with $a_{p/2}$ (or thereabouts) and the lower of the two is merged with $a_1$, $a_2$, . . . . $a_{p/2 - i}$ while the higher is merged with $a_{p/2 + 1}$, . . . . $a_{p - 1}$, $a_p$. Figure VI-5 shows $M_{10, 1}$ constructed this way as an example.

Another example, $M_{12, 4}$, is shown in Figure VI-6 with the subnetworks and sub-subnetworks identified by the dotted boxes. A proof that the above merging networks do in fact "merge" is given in Reference 2. (An $M_{m, n}$ network corresponds to the $M_{m, n}$ operator of this reference.)

Let $h(m, n)$ be the number of comparison elements in $M_{m, n}$. An exact expression for $h(m, n)$ is hard to obtain but an idea of how fast it grows is indicated by $h(2^p, 2^q - 2^p) = (p + 2)2^{q - 1} - 2^{p + 1} + 1$ (for $q \geq p \geq 0$). Other special cases of $h(m, n)$ are given in Reference 2.

As can be seen from Figure VI-4. doubling the size of $M_{m, n}$ adds one level of comparison elements to the network; hence, the longest path through the network is proportional to the logarithm of the size of the network; for example, the longest path in $M_{2^p, 2^q - 2^p}$ goes through q comparison elements.
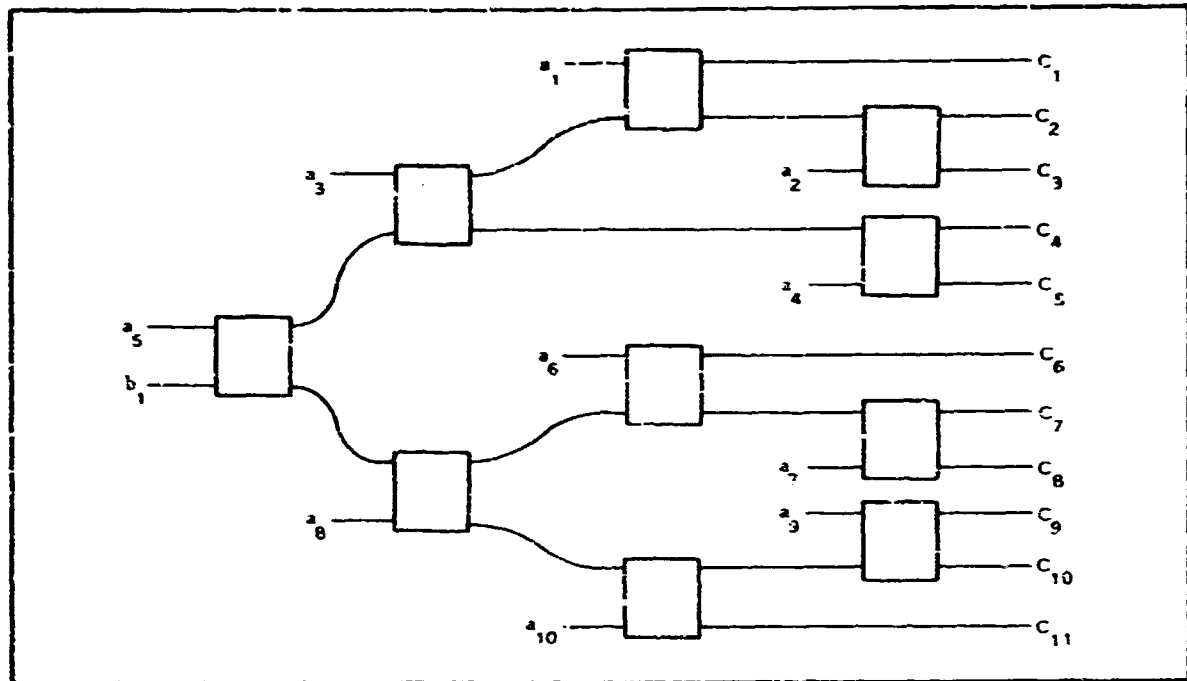
Figure VI-5 - $M_{10, 1}$ Merging Network

When the comparison elements include flip-flops for storage of data, there is a one clock-interval delay in each element and therefore it is necessary to add extra delays in the shorter paths of the network to equalize the delay in all paths. Delay elements or extra "waste" comparison elements can be used for this purpose. As an example, an $M_{2^p, 2^q - 2^p}$ network then would have q levels, with each level having $2^{q-1}$ comparison elements.

d. **Bi-Tonic Merging Networks ($N_{2^q}$)**

One disadvantage of $M_{m, n}$ is that it can merge only m items with n items and thus, for instance, cannot fulfill a need to merge m + 1 items with n - 1 items. There is another class of merging networks that has the capability of merging lists of items, regardless of the number of items in each list, subject only to the constraint that the total number of items to be merged is a power of two and remains constant.
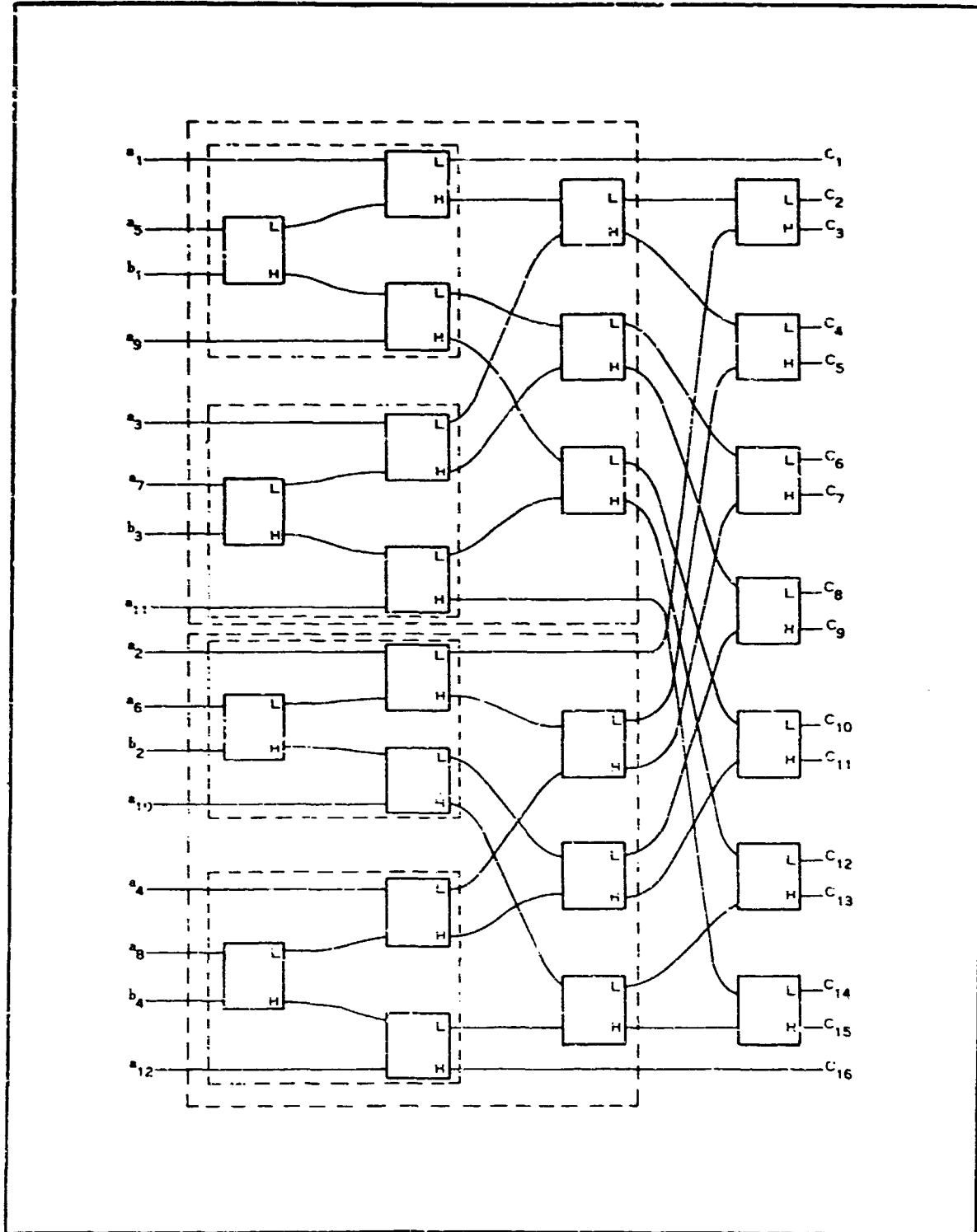
Figure VI-6 - $M_{12, 4}$ Merging Network

A sequence of numbers, $a_1$, $a_2$, . . . .,$a_{2^q}$ is bi-tonic if it is monotonic or if it consists of two monotonic sequences, one ascending the other descending, placed side by side (it does not matter whether the ascending sequence precedes the descending sequence or follows it). An $N_{2^q}$ bi-tonic merging network such as shown in Figure VI-7 can rearrange any bi-tonic sequence $a_1$, $a_2$, . . . , $a_{2^q}$ into an ascending sequence.

For any $q \geq 1$, $N_{2^q}$ can be constructed as follows. If $q = 1$, $N_2$ is simply one comparison element; if $q > 1$, $N_{2^q}$ consists of two $N_{2^{q-1}}$ networks and $2^{q-1}$ comparison elements connected as shown in Figure VI-8. A proof that these networks function as stated is given in Reference 3.

$N_{2^q}$ will have q levels and each level will have $2^{q-1}$ comparison elements. All paths traverse q elements so there is no need to add extra "waste" elements to equalize path lengths. To use $N_{2^q}$ as a merging network, one of the input sets should enter $N_{2^q}$ in ascending order and the other in descending order so that the total input set $a_1$, $a_2$, . . . , $a_{2^q}$ is a bi-tonic sequence.

e.   Sorting Networks

The merging networks of Items 3, c and 3, d above can be used to construct sorting networks by means of the well-known sorting-by-merging technique; for example, a network to sort $2^q$ items consists of $2^{q-1}$ comparison elements to arrange the items into $2^{q-1}$ ordered sequences of length q followed by $2^{q-2}$, $M_{2,2}$ or $N_4$ networks to merge these sequences into sequences of length 4, etc.

The total number of levels in a sorting network for $2^q$ items is 1/2 q(q + 1). If M-merging networks are used, the total number of elements is $(q^2 - q + 4)$ $2^{q-2} - 1$, while if N-merging networks are used the number of elements is $q(q + 1)2^{q-2}$.
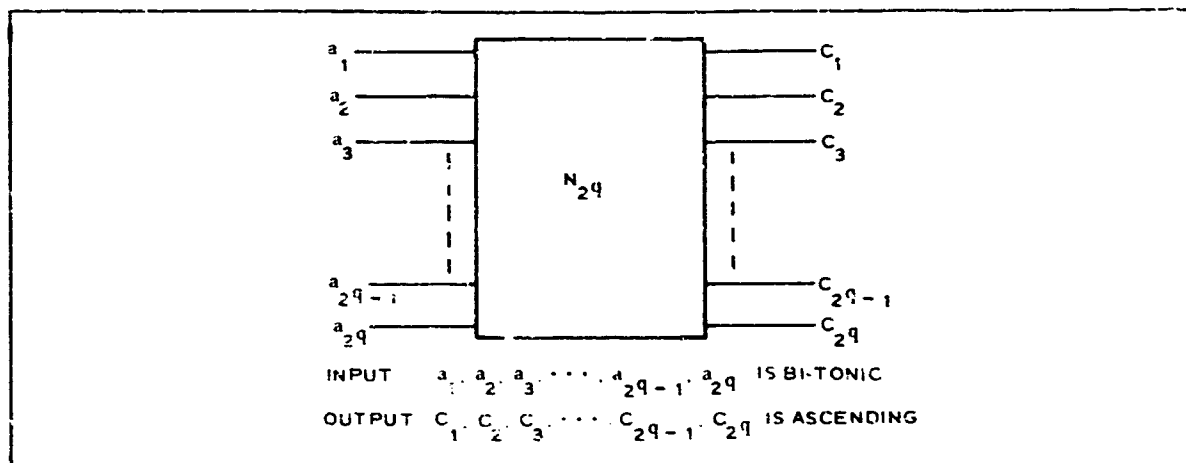
Figure VI-7 - Bi-Tonic Merging Network
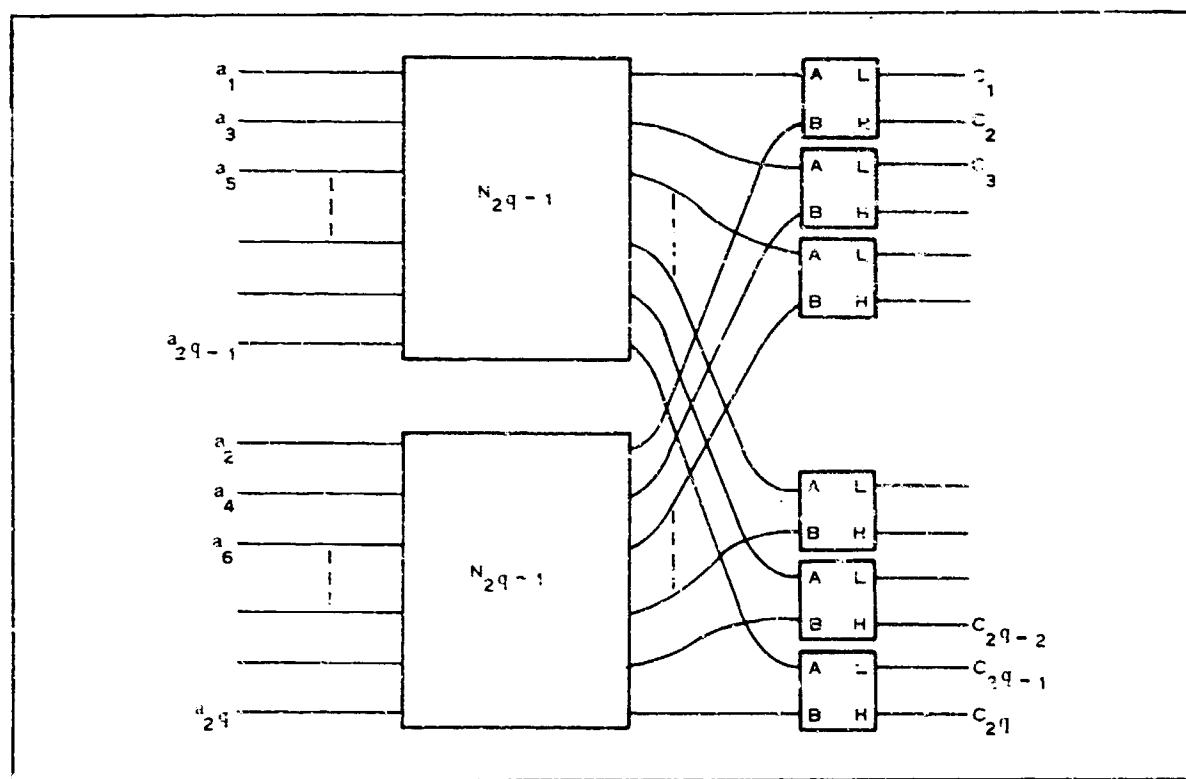


Figure VI-8 - Construction of $N_{2^q}$ from Two $N_{2^{q-1}}$ Networks and $2^{q-1}$

Comparison Elements

### 1. Separating Networks ($\widetilde{N}_{2^q}$)

A merging network combines two ordered sets of items into one ordered set. It is also desirable to have a network for the inverse operation; that is, a network that can separate an ordered set of items into two ordered sets (each item is marked with a flag bit to indicate the ordered set to which it belongs).

$\widetilde{N}_{2^q}$ (for $q \geq 1$) is defined as a bi-tonic separating network. That is, if $2^q$ items $d_1$, $d_2$, . . . , $d_{2^q}$ are presented over its inputs with $d_1 \leq d_2 \leq$ . . . $\leq d_{2^q}$ and if k of these items are flagged ($0 \leq k \leq 2^q$), then $\widetilde{N}_{2^q}$ presents the k flagged items on outputs $e_1$, $e_2$, . . . , $e_k$ ordered so that $e_1 \geq e_2 \geq$ . . . $\geq e_k$, and presents the $2^q - k$ unflagged items on $e_{k+1}$, $e_{k+2}$, . . . , $e_{2^q}$ ordered so that $e_{k+1} \leq e_{k+2} \leq$ . . . $\leq e_{2^q}$.

$\widetilde{N}_{2^q}$ can be constructed by an iterative process that is analogous to the process used for constructing $N_{2^q}$. Observe from Figure VI-8 that each element in the last level receives an item from the set $a_1$, $a_3$, $a_5$, . . . , $a_{2^q-1}$ on its A input and an item from the set $a_2$, $a_4$, $a_6$, . . . , $a_{2^q}$ on its B input. This suggests that each element in the first level of $\widetilde{N}_{2^q}$ should decide which of its two inputs needs an odd index in the final output, $e_1$, $e_2$, . . . , $e_{2^q}$, and which needs an even index.

From the definition of $\widetilde{N}_{2^q}$, the following rules (where $1 \leq i \leq 2^q$) can be established:

1. If i is even and if the set $\left\{ d_i, d_{i+1}, . . . . , d_{2^q} \right\}$ contains an even number of flagged items, then the item $d_i$ belongs in the set $E = \left\{ e_2, e_4, e_6, . . . . , e_{2^q} \right\}$.

    2.    If $i$ is even and if the set $\left\{d_i,\ d_{i+1},\ \ldots,\ d_{2^q}\right\}$ contains an odd number of flagged items, then the item $d_i$ belongs in the set $0 = \left\{e_1,\ e_3,\ e_5,\ \ldots,\ e_{2^q-1}\right\}$.

    3.    If $i$ is odd, then $d_i$ belongs in the complement to the set containing $d_{i+1}$; that is, $d_i \in E$ if $d_{i+1} \in 0$ and $d_i \in 0$ if $d_{i+1} \in E$

To establish these rules, any integer $i$ is considered as $1 \leq i \leq 2^q$. Let $t$ be the number of flagged items in $d_i,\ d_{i+1},\ \ldots,\ d_{2^q}$. From the definition of $N_{2^q}$, $d_i \longrightarrow e_t$ if $d_i$ is flagged and $d_i \longrightarrow e_{i+t}$ if $d_i$ is unflagged. Therefore, if $i$ is even, $d_i$ belongs to E or 0 depending on whether $t$ is even or odd, respectively. This establishes rules 1 and 2. The case of $i$ odd is divided into four subcases, all satisfied by rule 3:

    1.    If $d_i$ is flagged and $d_{i+1}$ flagged, then $d_i \longrightarrow e_t$ and $d_{i+1} \longrightarrow e_{t-1}$

    2.    If $d_i$ is unflagged and $d_{i+1}$ unflagged, then $d_i \longrightarrow e_{i+t}$ and $d_{i+1} \longrightarrow e_{i+t+1}$

    3.    If $d_i$ is flagged and $d_{i+1}$ unflagged, then $d_i \longrightarrow e_t$ and $d_{i+1} \longrightarrow e_{i+t}$

    4.    If $d_i$ is unflagged and $d_{i+1}$ flagged, then $d_i \longrightarrow e_{i+t}$ and $d_{i+1} \longrightarrow e_t$

An $\widetilde{N}_{2^q}$ network (Figure VI-9) can be built from two $\widetilde{N}_{2^{q-1}}$ networks preceded by a set of separating elements. An $\widetilde{N}_2$ network is simply one separating element. Each of the separating elements in the first level of $\widetilde{N}_{2^q}$
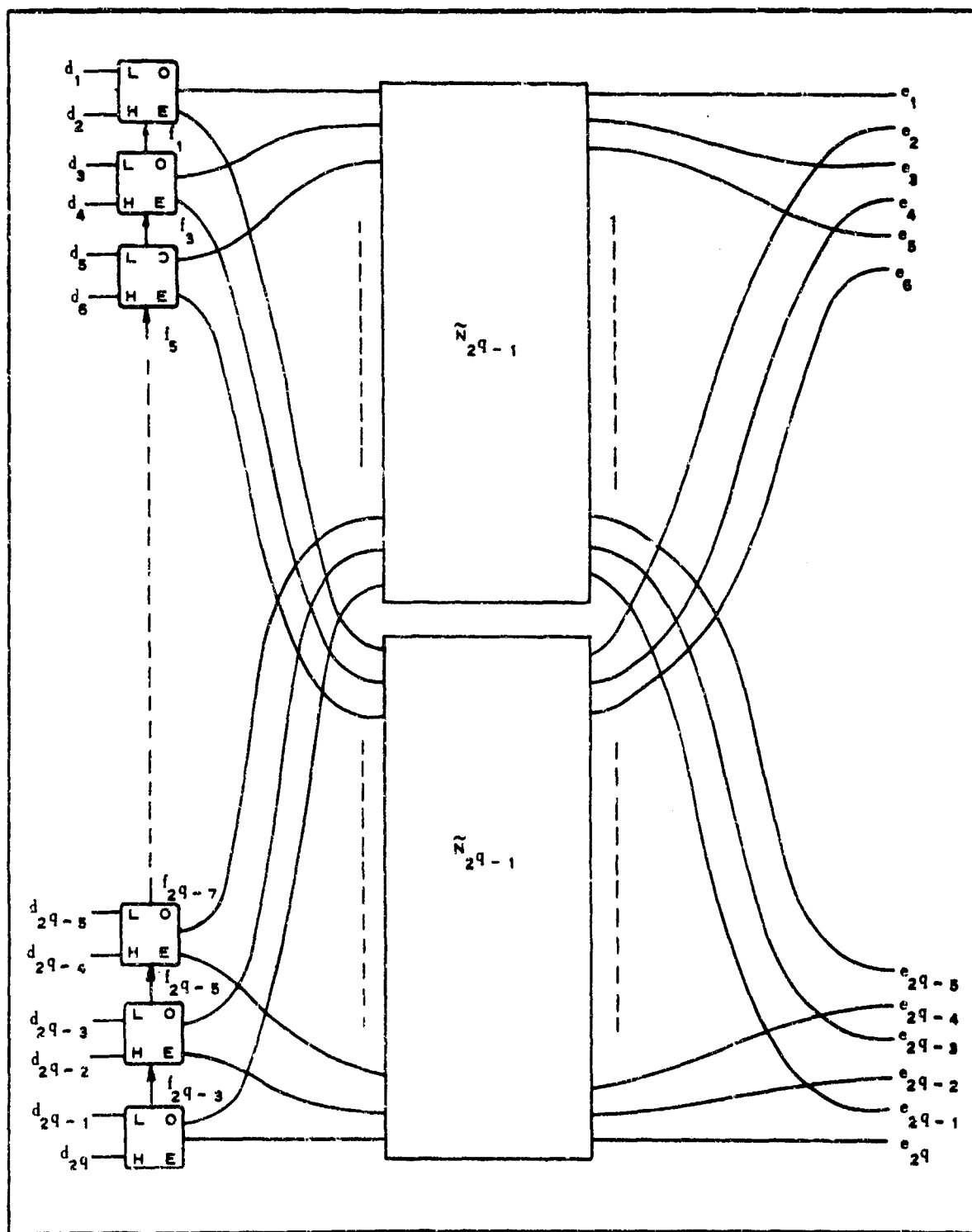
Figure VI-9 - Construction of $\tilde{N}_{2^q}$

receives two items of data ($d_i$ and $d_{i+1}$) over its L and H inputs (the first bit received in each item is its flag bit). It also receives an indication ($f_i$) of whether the number of flagged items in the set $\left\{d_{i+2}, d_{i+3}, \ldots, d_{2q}\right\}$ is odd or even. From rules 1, 2, and 3 it decides whether $d_i \in O$ and $d_{i+1} \in E$ or $d_i \in E$ and $d_{i+1} \in O$ and presents $d_i$ and $d_{i+1}$ on its O and E outputs accordingly.

The $f_i$ signals are generated in a network of exclusive-or circuits that receive the flag bits. Long ripple times can be avoided by using a "look-ahead" structure similar to the carry-look-ahead structures of MacSorley.[4]

### g. A Multiaccess Memory Using Merging and Separating

The networks described in Items 3, c, d, e, and f above can be combined to form a multiaccess memory (Figure VI-10). The elements in the $N_{2^q}$ and $\widetilde{N}_{2^q}$ networks incorporate shift-register stages that store the bits of data in memory. The memory words recirculate the most-significant bits first through the $N_{2^q}$ and $\widetilde{N}_{2^q}$ networks via the paths a, b, and c in Figure VI-10. Each memory word has an address field in its most-significant portion followed by a data field followed by a control field of $p + 2$ ONEs. The words in memory are kept in ascending order; that is, the word with the highest address field is at the top of memory, etc. Words with equal address fields are in adjacent locations ordered by their data fields (in order that words be in algebraic order, a ONEs or TWOs complement system is used with the sign-bit complemented). Empty words have zeroes in all digits so that they fill the bottom portion of memory. The order is kept in memory by sorting all new words each cycle and merging them with the memory words in $N_{2^q}$.

There are q levels in $N_{2^q}$, q levels in $\widetilde{N}_{2^q}$, and one level in the transfer network so the recirculation paths have a delay of $2_{q+1}$ clock pulses. The word length is a multiple of $2q + 1$ and transmission is done in serial-parallel form. Words will recirculate twice per basic memory cycle and
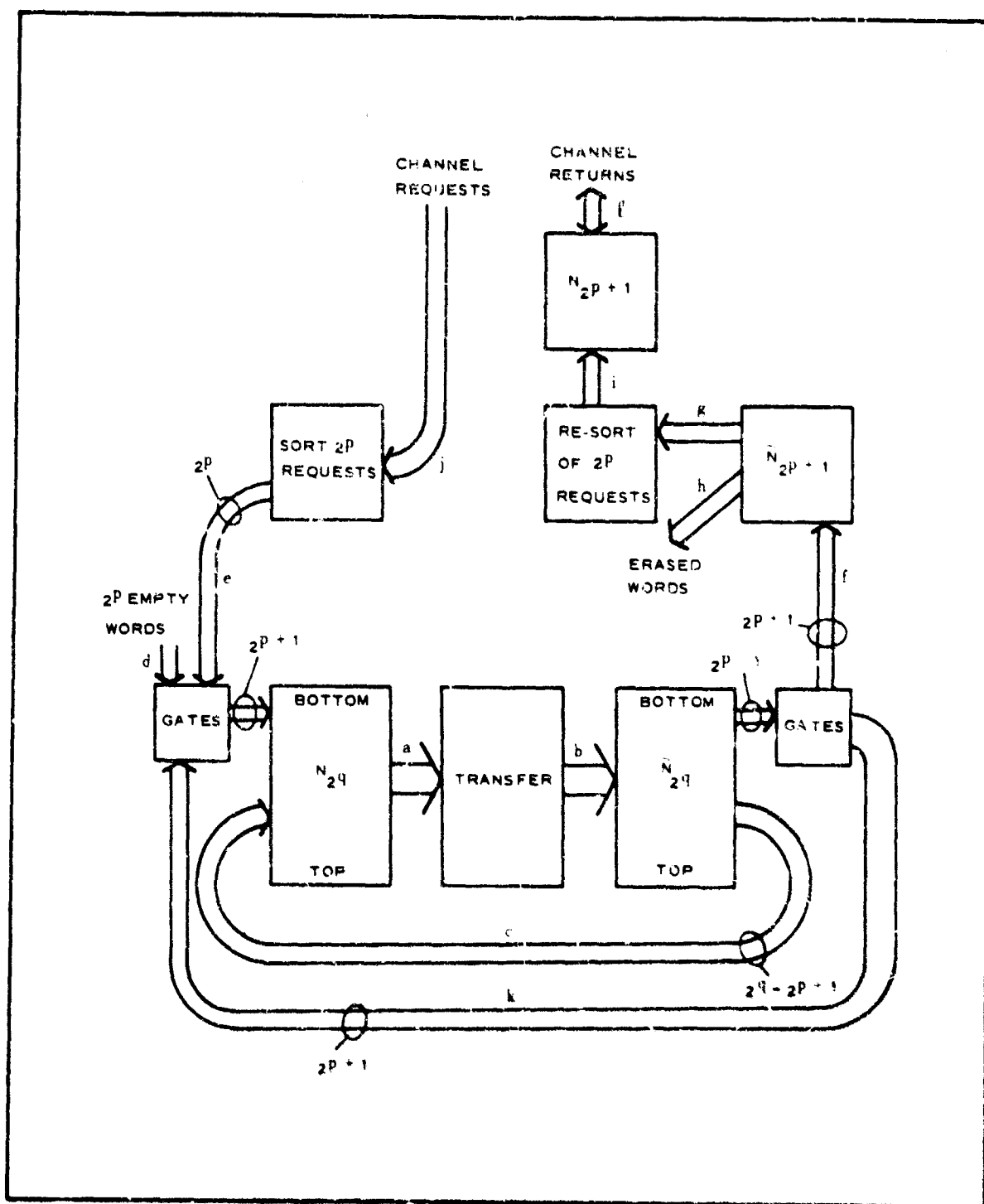
Figure VI-10 - A Multiaccess Memory with $2^q - 2^p + 1$ Words and $2^p$ Requests

thus the memory cycle time is $4q + 2$ clock pulses. In operation, requests arrive over the $2^p$ input channels (j in Figure VI-10). These have the same format as the memory words. There are three types of requests: WRITE, READ, and READ AND ERASE. A WRITE request has ones in its control field and has the address and data of the word to be written in its address and data fields. A write request does not overwrite old data but rather creates a new word. A READ request has the bits 01xx . . . xx in its control field where xx . . . xx is the channel number. A READ AND ERASE request has the bits 10xx . . . xx in its control field where xx . . . xx is the channel number. The address and data fields of a READ or READ AND ERASE request indicate the word to be read. If there is a word in memory whose address and data fields agree with that of the request, that word is read; otherwise, the memory word that is immediately higher than the request is read. As examples, if the data field of a request is all zeros and if there is one memory word whose address agrees with the request, that memory word is read; if several words have addresses agreeing with the request, the one with the least data field is read, etc. A READ AND ERASE request will erase a memory word after reading it. Any inactive channel will have a request word of all zeros.

A sorting network orders the requests with the address field taking precedence over the data field, etc. The ordered requests are presented to $N_{2^q}$ over the e lines (Figure VI-10) with the highest request entering the bottom of memory, the next lower request entering the next higher word, etc. Concurrently, the memory words enter $N_{2^q}$ over c in ascending order. The $2^p$ empty words enter on the d lines. The input to $N_{2^q}$ is a bi-tonic sequence (Item 3, d above) and therefore q clock pulses later the merged requests and memory words start leaving $N_{2^q}$ in ascending order.

Merging continues until the control field starts to enter the transfer network. Meanwhile, the address and data fields circulate through $\tilde{N}_{2^q}$ and

back into $N_{2^q}$ via the k and c lines. The $\widetilde{N}_{2^q}$ network is set so that no reordering of the words occurs. Also, the comparison elements of $N_{2^q}$ are reset as the words re-enter $N_{2^q}$ so that the ascending order is preserved.

At this point in the memory cycle, the first two bits of each control field are residing in the transfer network. These bits are 00 for empty words, 01 for READ requests, 10 for READ AND ERASE requests, and 11 for memory words. The transfer network remembers this information and changes these two bits to the code:

01 for unerased memory words
10 for erased memory words and empty words
11 for READ and READ AND ERASE requests

The code change is mechanized easily. An erased memory word can be distinguished from an unerased memory word since it is located immediately above a READ AND ERASE request. The first bit of the new code will be a flag for $\widetilde{N}_{2^q}$ to indicate the words to be removed from memory.

During the next few clock pulses the remainders of the control fields are fed through the transfer network with no change. As the address and data fields are fed through the transfer network, the address and data fields of each READ and READ AND ERASE request are overwritten with the address and data fields of the first memory word above the request.

As the words proceed through $\widetilde{N}_{2^q}$, the flagged words are separated from the unflagged and exit at the bottom of $\widetilde{N}_{2^q}$. The $2^q - 2^{p+1}$ topmost words enter $N_{2^q}$ again via the c lines and a new memory cycle starts.

The $2^{p+1}$ bottommost words contain all requests, all erased words and some empty words. They are put into $\widetilde{N}_{2^{p+1}}$ via the f lines. $\widetilde{N}_{2^{p+1}}$

uses the second control bit as a flag to separate requests from erased words.

The requests enter a sorting network (now with the control field in the most-significant place), which resorts the requests by channel numbers. The sorted requests enter an $N_{2^p + 1}$ network (via i) along with the channel numbers (via $\ell$). In this network, the comparison elements contain reverse paths along with forward paths. The comparisons are performed on numbers in the forward paths but the switching action of each element affects both the forward paths and the reverse paths. By this means, the request can be fed back on the correct channel via $\ell$.

Assuming logic elements with 10-nsec propagation delays, a 30,000-word memory should have a memory cycle time of 8 μsec and an access time of 16 μsec (the access time is longer than the cycle time because of the time spent in sorting and re-sorting the requests and because it includes the time to transmit the full requested word). These times assume 1024 request channels. Because 1024 requests can be processed every 8 μsec, an effective cycle time of 8 nsec is obtained.

4.   PARALLEL COMPUTER ORGANIZATION

The multiaccess self-sorting memory of Item 3, g above can be used in a parallel computer organization. An example parallel processor might have 300 arithmetic units, each with its own accumulator, quotient register, index registers, instruction register, program counter, and request channel to the memory. Because these arithmetic units can be mass produced, it is expected that each would be considerably cheaper than an arithmetic unit of a normal computer.

The instruction set of each arithmetic unit is similar to that of a normal computer: LOAD A, LOAD Q, STORE A, STORE Q, ADD, SUB, MULTI- PLY, DIVIDE, SHIFT, JUMP, etc. There are a few differences, however:

1. Each instruction that reads an operand from memory
   has two modifying bits. One bit indicates whether
   the operand should be erased from memory or not;
   that is, whether the read request sent to memory
   should be READ or READ AND ERASE. The other
   bit indicates whether the address of the operand
   accepted from memory should agree with the re-
   quested address or not; that is, since the memory
   always returns an operand for each read request,
   the operand address will be different from the re-
   quested address if no word in memory has the re-
   quested address. If this occurs, the second bit in-
   dicates whether to reinitiate the request or whether
   to use the operand received from memory. Some
   cases require both modes.

2. Each store instruction creates a new word in memo-
   ry instead of overwriting an old word. This makes
   it possible to store several words in memory with
   the same address (a set of words can be ordered
   simply by giving each item the same address).

3. Most instructions that read an operand from memo-
   ry will fetch the minimum item if there is more than
   one item with the same address. This is because
   the data field of the READ request to memory con-
   tains zero. It is also useful to be able to fetch a
   word in the middle of a list of items stored with the
   same address. Thus, there should be some fetch
   operations that use the contents of the accumulator
   for the data field of the read request. These oper-
   ations are similar to threshold searches in a nor-
   mal computer, such as the CDC-1604, except that

they require only one memory access time for execution regardless of the length of the table being searched.

4. Indirect addressing capability will be useful. Some programs can be executed faster if one arithmetic unit computes "addresses" while another refers to these by indirect addressing.

5. Indexable jumps are useful since there will be cases where several arithmetic units may be executing the same subroutine and the return addresses have to be stored in the arithmetic units themselves.

6. Interrupting capability is useful so one "master" arithmetic unit can control the others easily. It also allows an arithmetic unit that is waiting for data to be interrupted and started on a new program.

7. A "skip" on the presence or nonpresence of an address in memory is useful for synchronizing arithmetic units.

The example processor might have hundreds of I/O channels, each with its own request channel to memory. There may also be a large backup store to main memory that uses several request channels so that large blocks of data can be moved in and out in parallel. With a large, fast-access backup store, a large main memory is not needed. The I/O equipment can be controlled by reserving certain addresses for I/O control-word storage.

This example parallel processor will be able to perform a large class of programs very fast. The sorting and searching capability allows fast data retrieval while the parallel arithmetic units allow programs to be executed in parallel. For example, in the processing of a list structure, the structure can be gone through in parallel, with a different arithmetic unit processing each branch and subbranch, etc. The parallel I/O channels allow fast data input and output, multiconsole arrangements, fast-access backup

-160-

stores, etc. With error-detecting capability in the arithmetic units and interrupt, it is possible to bypass failed arithmetic units without halting computation, greatly reducing machine down time.

5. CONCLUSIONS

This report shows how a fast multiaccess self-sorting memory can be constructed. It also gives an example of a parallel processor organization (Machine I) using this memory. Besides the parallel arithmetic capability, this organization has the following features:

1. The sorting capability in memory allows fast sorting and table searching.

2. The parallel I/O channels allow fast data input and output.

3. The organization utilizes content-addressing.

4. It is possible to bypass failed arithmetic units without halting computation. This will result in greatly reduced downtime.

5. A full complement of arithmetic units are not required for operation of the machine. Additional arithmetic units can be added later without changing the programs. As the number of arithmetic units increase the machine time (assuming sufficient parallelism in the program) will decrease.

6. The programmer does not need to assign tasks to each arithmetic unit. A list of tasks to be performed is stored in memory, each with the same address. The arithmetic unit(s) take the top item(s) from this list.

The organization of the example parallel processor may be modified if

programming studies indicate the need for other features. The merging networks in this paper may have an application in any communication switching problem. They can be made to resemble large crossbar switches but they have fewer elements.

6.  REFERENCES

1.  Slotnick, D. L., et al: <u>Solomon</u>. Proceedings of the Fall Joint Computer Conference, 1962.

2.  GER-11759: <u>A New Internal Sorting Method</u>. Akron, Ohio, Goodyear Aerospace Corporation, 29 September 1964.

3.  GER-11869: <u>Bi-Tonic Merging</u>. Akron, Ohio, Goodyear Aerospace Corporation, December 1964.

4.  MacSorley, O. E.: "High-Speed Arithmetic in Binary Computers." <u>Proceedings of the IRE</u>, January 1961, vol 49, no. 1.

# APPENDIX VII - PARALLEL MERGING-SEPARATING MEMORIES

## 1. INTRODUCTION

In a multiprocessor using a sorting memory as a multiaccess memory, the full sorting capability of the memory is not needed since most of the memory words remain in the same order from one cycle to the next. Only new additions, read requests, and erasures cause changes and these are a small fraction of all the words in memory. This leads to the concept of using a merging-separating memory in place of a complete sorting memory. The cycle time of a sorting memory of $2^n$ words is $1/2n (n + 1)$ steps while that for a merging-separating memory of $2^n$ words is $2n + 2$ steps. Thus, a faster cycle time should be expected in a merging-separating memory (some of its steps will be longer but there still will be a time advantage).

## 2. FUNCTIONAL DESCRIPTION OF A MERGING-SEPARATING MEMORY

A merging-separating memory cycle has four phases: merging, flagging, separating, and exchanging (see Figure VII-1). At the beginning of a merging phase, the set of memory words is divided into two parts, the higher containing the words of memory left from previous cycles arranged in numerical order, and the lower containing new additions and read requests arranged in order. In the merging phase, these two parts are merged so the old memory words, new additions, and read requests form one ordered list in memory. In the flagging phase, the contents of the requested memory words are transferred to the read requests, the read requests are flagged, and the memory words to be erased (which are associated with read and erase requests) are flagged.

In the separating phase, the flagged words are separated from the unflagged, the unflagged memory words left in the higher part of memory

are arranged in order, and the flagged read requests and erasures left in the lower part are arranged in order. In the exchanging phase, equipment external to the memory (not shown in Figure VII-1) reads the read requests and erasures and replaces the lower part of memory with a new ordered list of new additions and read requests for the next cycle.

The merging and separating phases are most easily realized if the lower lists are arranged in an order opposite that of the upper lists. A bi-tonic merge performs the merging and the separating is done by an "inverse" network (see Appendix VI).

Let the $2^n$ words of the memory be indexed with 0, 1, 2, . . . ., $2^n - 1$ with 0 the index of the word at the low end and $2^n - 1$ the index of the word at the high end. In each step of the merging phase, the $2^n$ words are formed into $2^{n-1}$ pairs. The two words in each pair are compared and if the word with the lower index is higher in magnitude than the word with higher index, the two words are exchanged; otherwise, the pair is left alone.

The pairing rule is explained easily if the indices are considered as written in binary form, the bits of the indices are indexed by 1, 2, 3, . . . ., n (1, the most-significant bit and n, the least-significant bit), and the steps of merging phase are indexed 1, 2, . . . ., n in sequential order. The pairing rule is: "On step k, word i is paired with word j if and only if bit k of word i is not equal to bit k of word j and all other corresponding bits of i and j are equal. "

As an example, in a 16-word merging-separating memory, the first merging step treats the following eight pairs:

(0, 8) (1, 9) (2, 10) (3, 11) (4, 12) (5, 13) (6, 14) (7, 15).

The second merging step treats these pairs:

(0, 4) (1, 5) (2, 6) (3, 7) (8, 12) (9, 13) (10, 14) (11, 15).

The third merging step treats these pairs:

-164-

$$(0,2)\ (1,3)\ (4,6)\ (5,7)\ (8,10)\ (9,11)\ (12,14)\ (13,15).$$

The fourth merging step, these pairs:

$$(0,1)\ (2,3)\ (4,5)\ (6,7)\ (8,9)\ (10,11)\ (12,13)\ (14,15).$$

In each step of the separating phase, the $2^n$ words also are formed into $2^{n-1}$ pairs. The pairing rule is the reverse of that for the merging phase; that is, the first separating step works with the same pairs as those considered in the last merging step, the second separating step corresponds to the next-to-the-last merging step, etc. The two words in each pair may or may not be exchanged. The exchange rule is explained as follows. For

$$0 \le i \le 2^n - 1 \text{ and } 0 \le k \le n - 1,$$

let

$$S_{i,k} = \left\{ j:\ i < j \le 2^n - 1 \right\}$$

and the least-significant k bits of j equal the corresponding least-significant k bits of i.

As an example, if $n = 3$, then

$$S_{0,0} = \{1,2,3,4,5,6,7\} \quad S_{0,1} = \{2,4,6\} \quad S_{0,2} = \{4\}$$
$$S_{1,0} = \{2,3,4,5,6,7\} \quad S_{1,1} = \{3,5,7\} \quad S_{1,2} = \{5\}$$
$$S_{2,0} = \{3,4,5,6,7\} \quad S_{2,1} = \{4,6\} \quad S_{2,2} = \{6\}$$
$$S_{3,0} = \{4,5,6,7\} \quad S_{3,1} = \{5,7\} \quad S_{3,2} = \{7\}$$
$$S_{4,0} = \{5,6,7\} \quad S_{4,1} = \{6\} \quad S_{4,2} = \phi$$
$$S_{5,0} = \{6,7\} \quad S_{5,1} = \{7\} \quad S_{5,2} = \phi$$
$$S_{6,0} = \{7\} \quad S_{6,1} = \phi \quad S_{6,2} = \phi$$
$$S_{7,0} = \phi \quad S_{7,1} = \phi \quad S_{7,2} = \phi$$
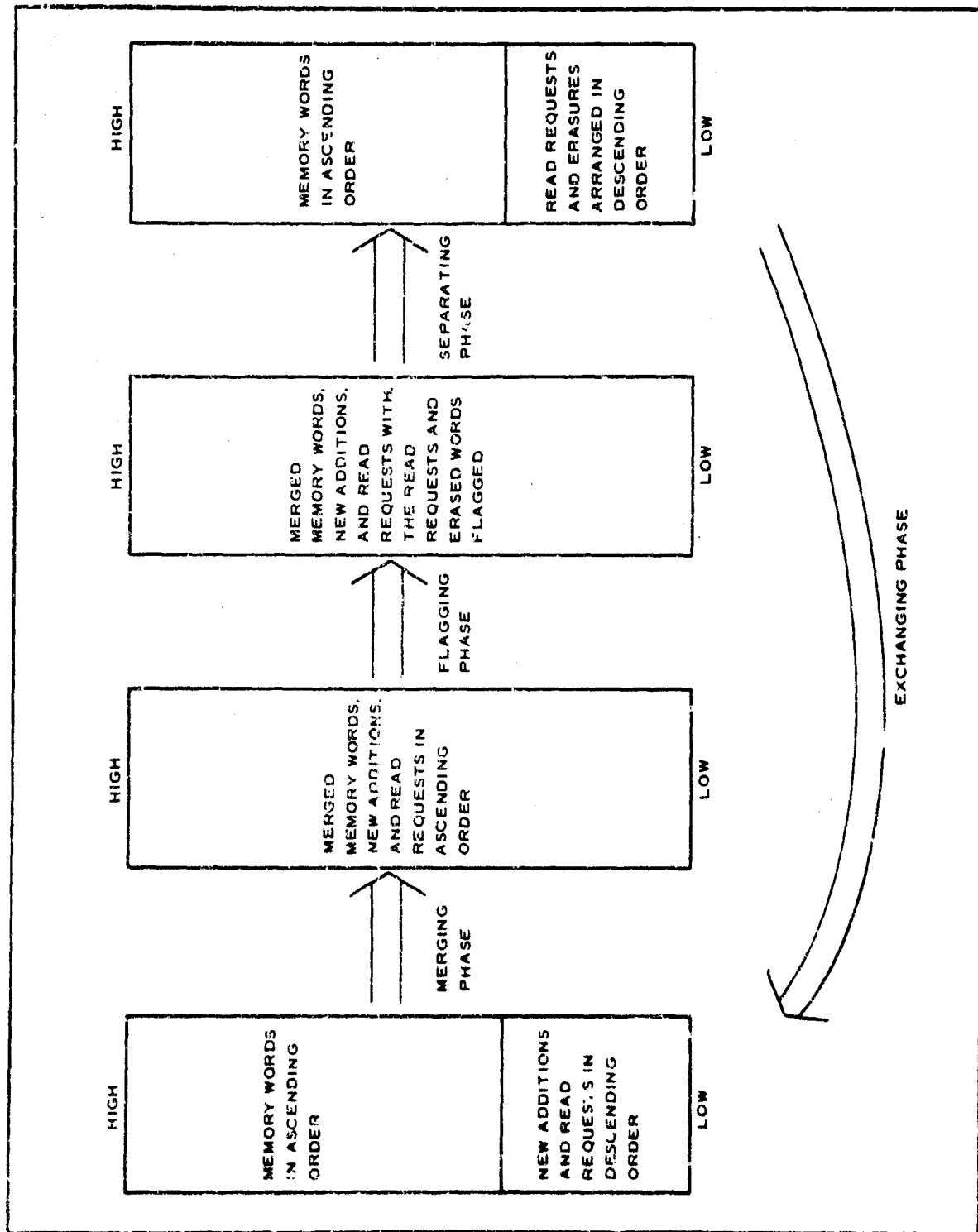
Figure VII-1 - Merging-Separating Memory Cycle

where $\phi$ denotes an empty set. Let $t_{i, k}$ denote the number of flagged words with indices in $S_{i, k}$ (if $S_{i, k}$ is empty, $t_{i, k} = 0$). The exchange rule for the pair of words with indices $m$ and $m + 2^{k-1}$ at separating step $k$ is: "Exchange $m$ and $m + 2^{k-1}$ if and only if $t_{m, k-1}$ is odd."

As an example, suppose words 3, 6, and 7 are flagged in an eight-word memory. Then $t_{0, 0}$, $t_{1, 0}$, $t_{2, 0}$, and $t_{6, 0}$ are odd and $t_{3, 0}$, $t_{4, 0}$, $t_{5, 0}$, and $t_{7, 0}$ are even. In the first separating step, the words in pairs (0, 1), (2, 3), and (6, 7) are exchanged and the words in pair (4, 5) are unchanged. The flagged words are now in 2, 6, and 7 so $t_{1, 1}$, $t_{2, 1}$, $t_{3, 1}$, $t_{4, 1}$, and $t_{5, 1}$ are odd and $t_{0, 1}$, $t_{6, 1}$ and $t_{7, 1}$ are even. In the second separating step, the words in pairs (1, 3), (4, 6), and (5, 7) are exchanged and the words in pair (0, 2) are unchanged. The flagged words are now in 2, 4, and 5 so $t_{0, 2}$ and $t_{1, 2}$ are odd, $t_{2, 2}$, $t_{3, 2}$, $t_{4, 2}$, $t_{5, 2}$, $t_{6, 2}$ and $t_{7, 2}$ are even. In the third separating step, the words in pairs (0, 4) and (1, 5) are exchanged and the words in pairs (2, 6) and (3, 7) are unchanged. Figure VII-2 shows the interchanges performed (underlined indices indicate the flagged words). In this example, the three flagged words were moved to the low end of memory with their order reversed and the unflagged words were moved to the high end with their order preserved.

## 3. PARALLEL MEMORY

In the parallel form of a merging-separating memory, there are half as many word stores as words. Each word store contains storage for two words plus the logic for the comparing, flagging, and separating functions.

In the n merging + n separating steps of a $2^n$-word memory, words shift between the word stores so that each of the desired pairs is formed. The words can be arranged in memory so that the same wires can be used between each pair of consecutive steps of the merge. An example is shown in Figures VII-3 and VII-4. In Figure VII-3, the eight pairs in each of the
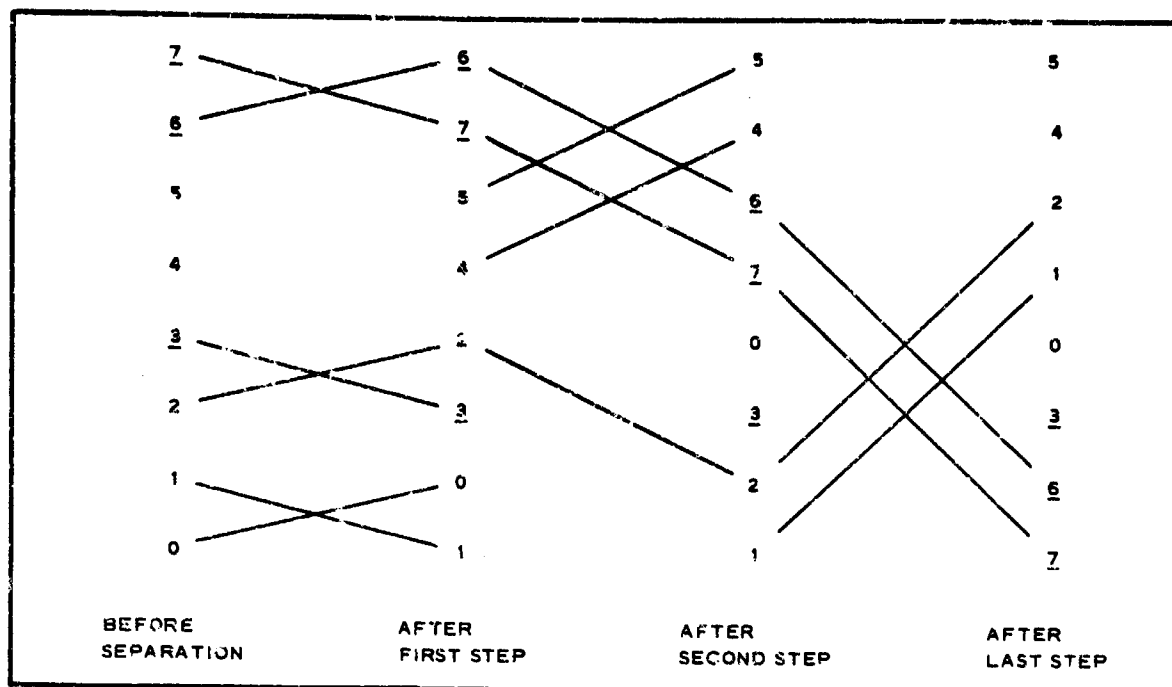
Figure VII-2 - Example of Word Interchanges

four steps of a 16-word merge are arranged so that the wiring patterns between all pairs of consecutive steps are identical; in Figure VII-4, $\circ$ elements are shown interconnected so that they can be used to perform the same function as the 32 elements of Figure VII-3.

If the words in a $2^n$-word memory are indexed by 0, 1, 2, . . ., $2^{n-1}$ and if each location is given the same index as the word it contains in the last merging step, then the location of any word at any step is given by the rule: "Word i is in location j at step k if the n-bit binary representation of j is the binary representation of i shifted right end-around n - k places."

The wiring rule between the locations is: "Location i feeds location j if the n-bit binary representation of j is the same as that for i shifted left one place end-around." This shows the wiring necessary for the merging phase.

Since the steps of the separating phase are in the reverse order, the wires
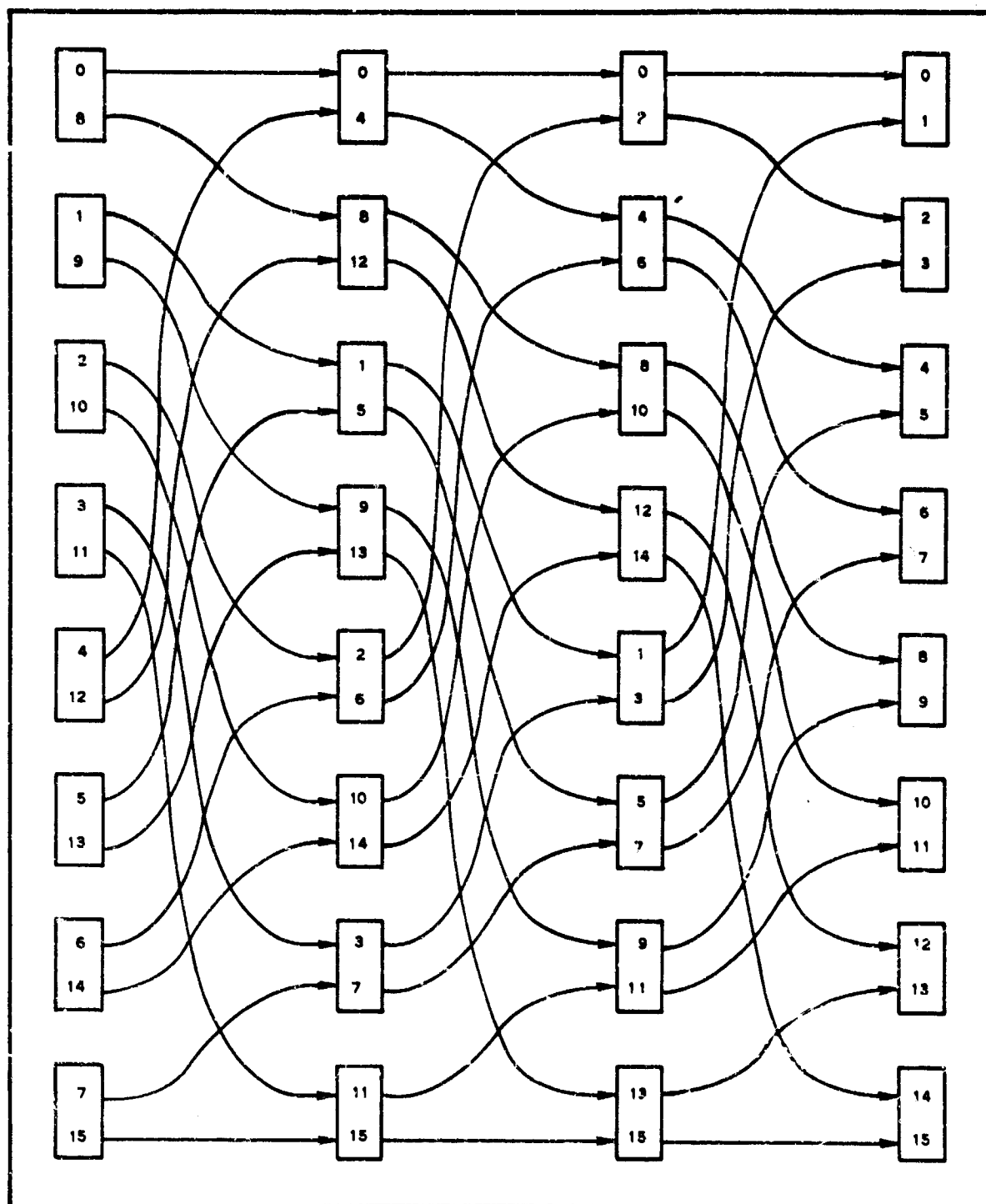
Figure VII-3 - Sixteen-Word Merge Arranged for Same Wiring
Pattern between Each Pair of Levels

-169-
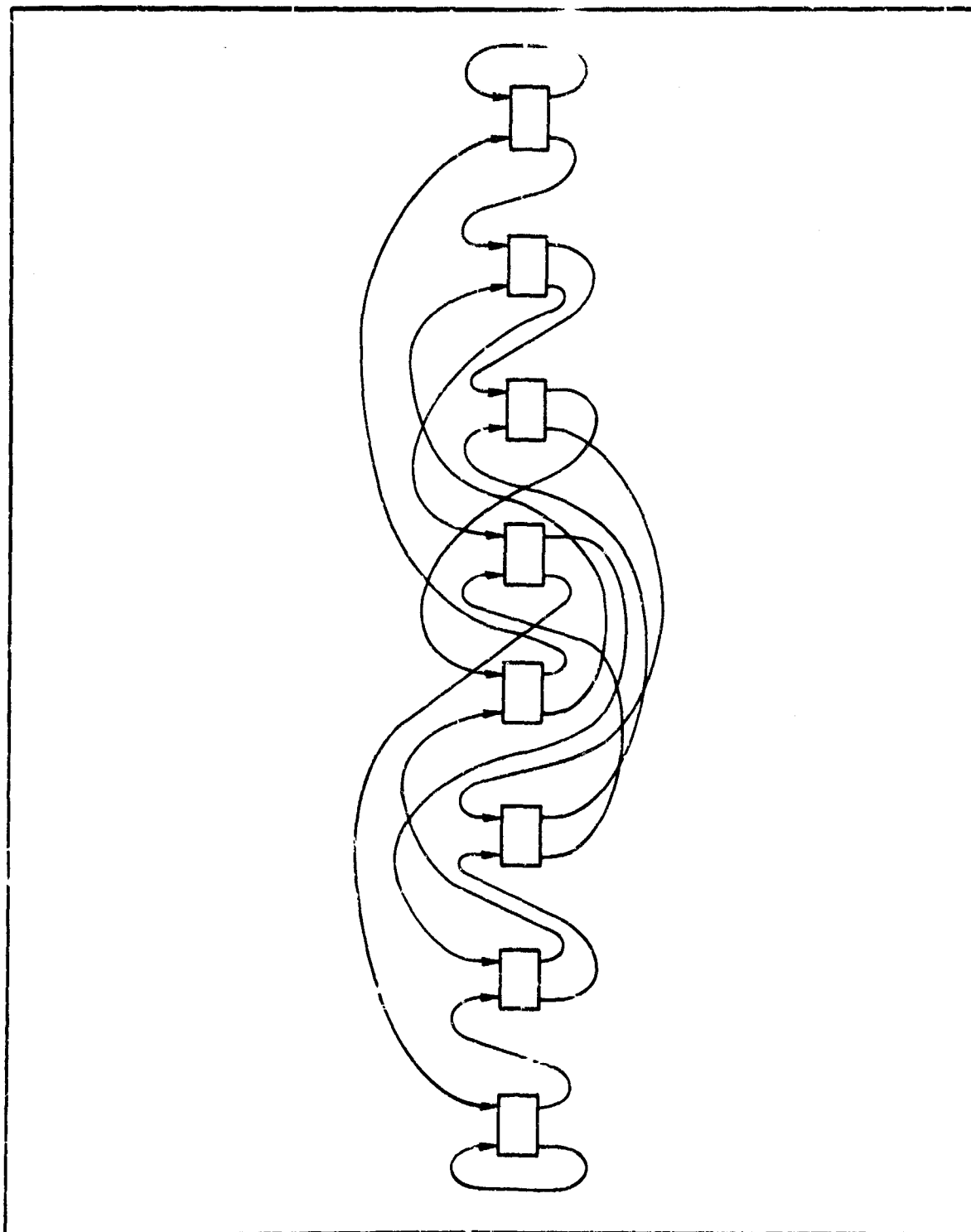
Figure VII-4 - Eight-Element Version of 16-Word Merge

NOTES:

1. M CONTROL SIGNAL IS "ON" DURING THE MERGING PHASE.

2. S CONTROL SIGNAL IS "ON" DURING THE SEPARATING PHASE.

3. EXAMPLE ASSUMES i FEEDS j IN THE MERGING PHASE.
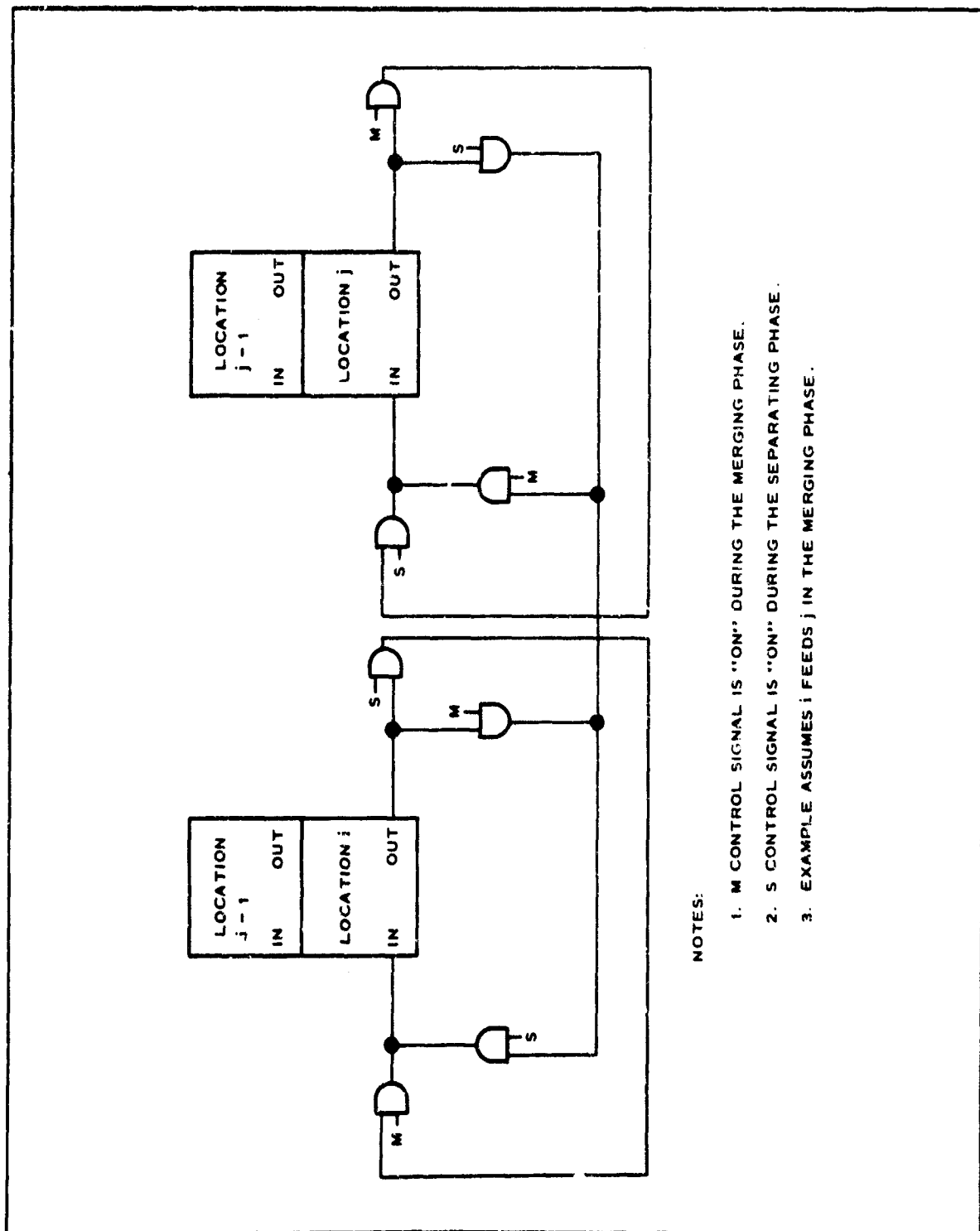
Figure VII-5 - Example of Use of Same Wires for Merging and Separating
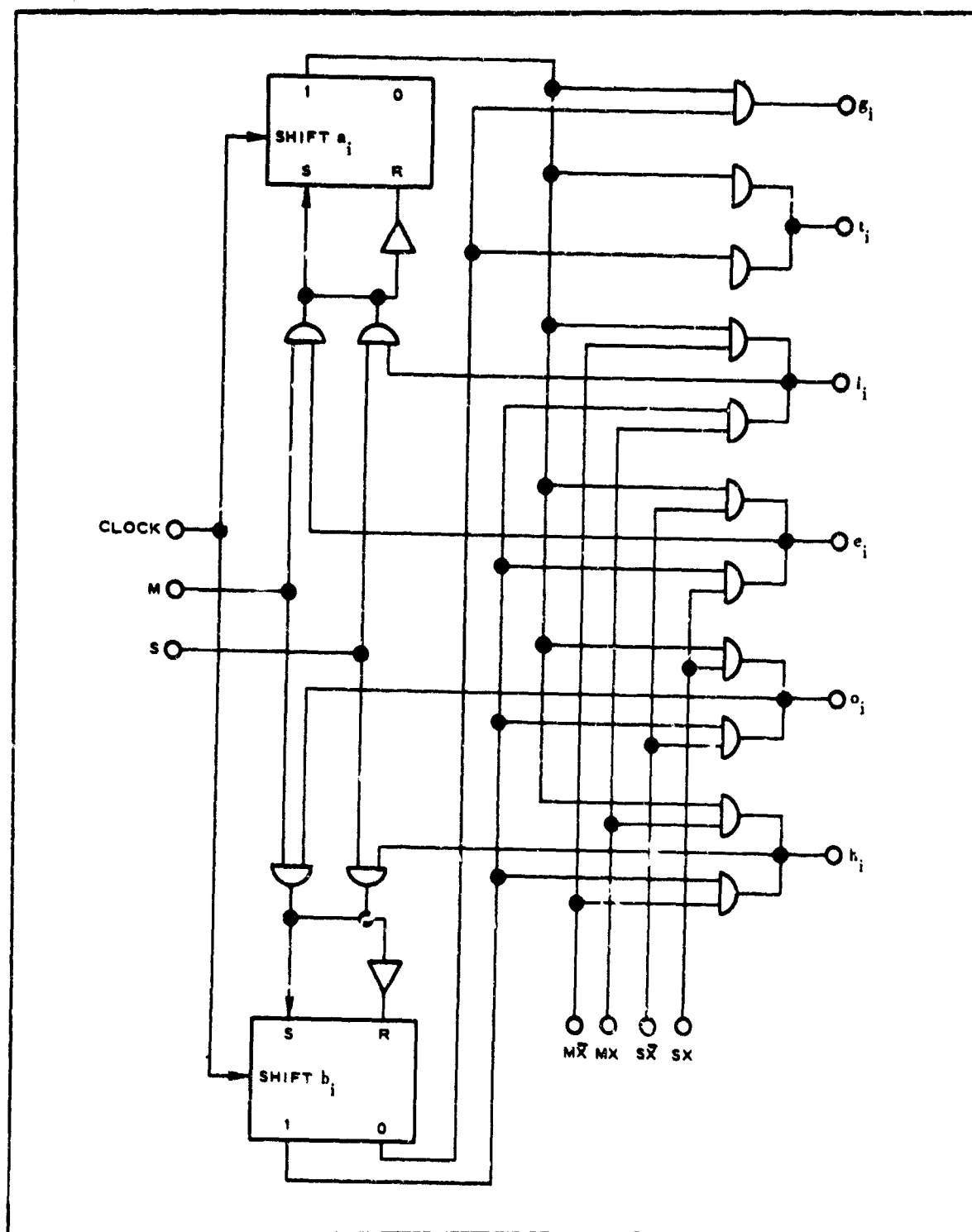
Figure VII-6 - Digit Store ($i^{th}$ Digits in an Element)

$$T = \bigwedge_{i=1}^{p} t_i$$

$$G = \bigvee_{j=1}^{p} g_j \bigwedge_{i=j+1}^{p} t_i$$
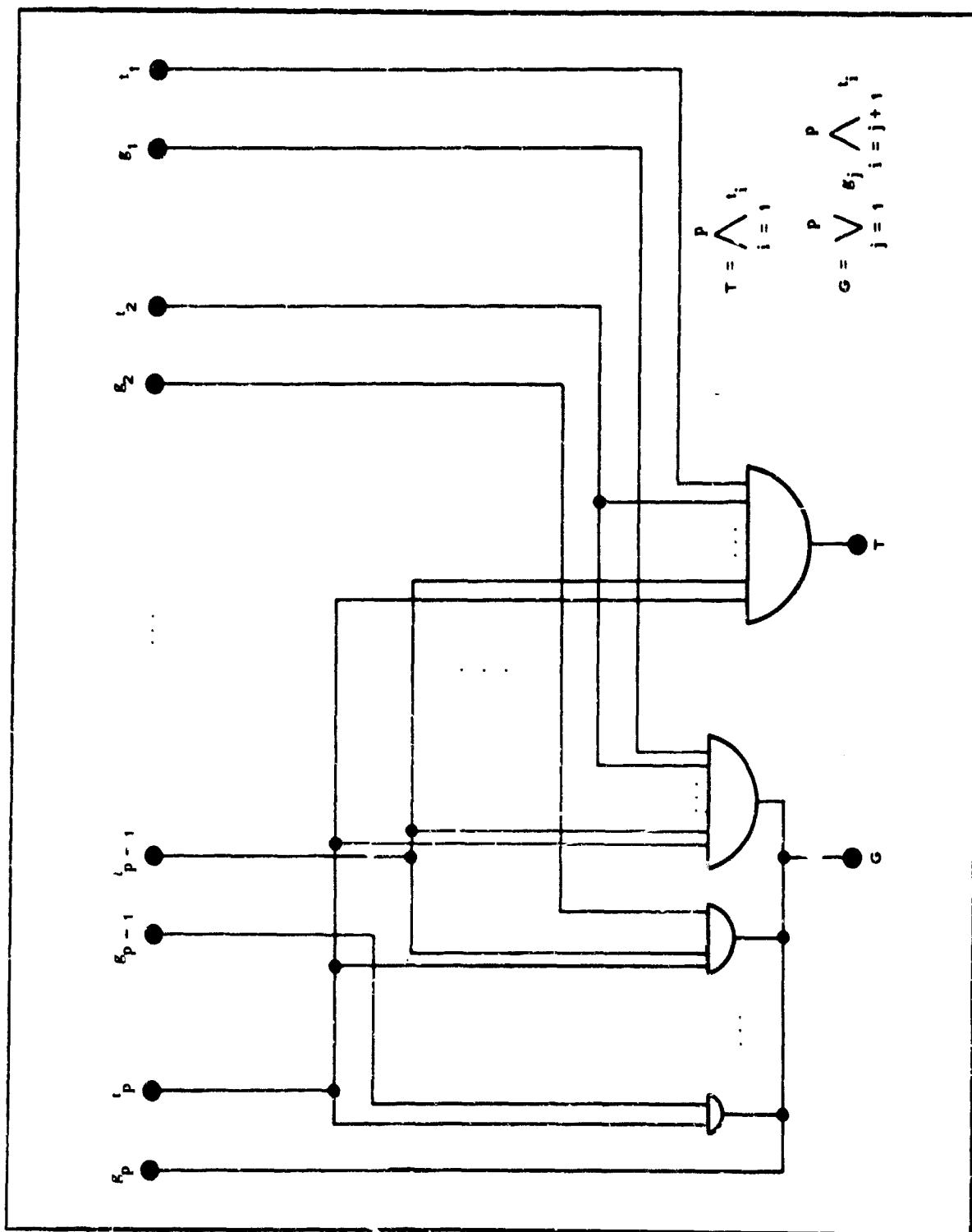
Figure VII-7 - p-Group for the Comparison Circuit
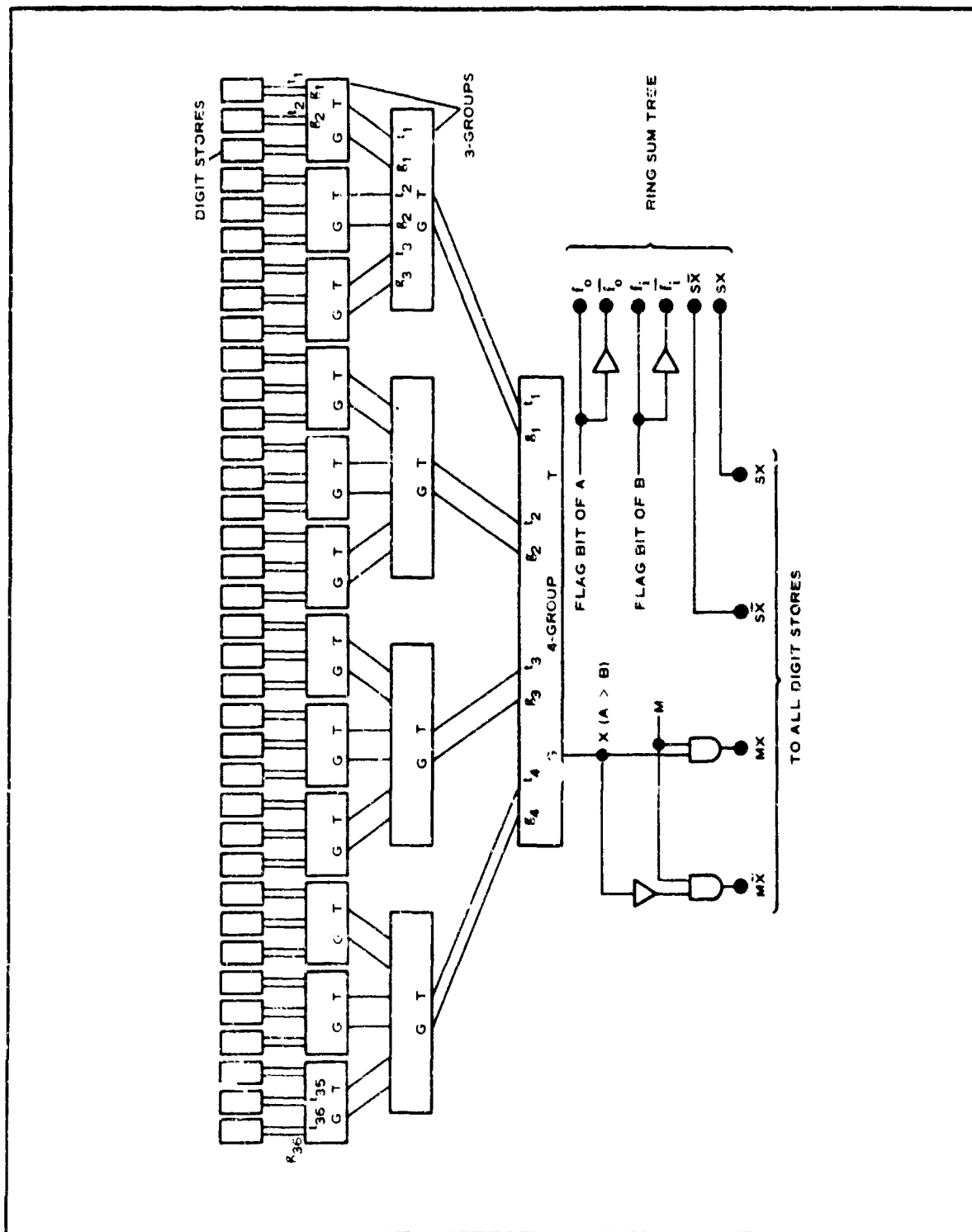
Figure VII-8 - Word Store for 36-Bit Words (3-Level Cascade)
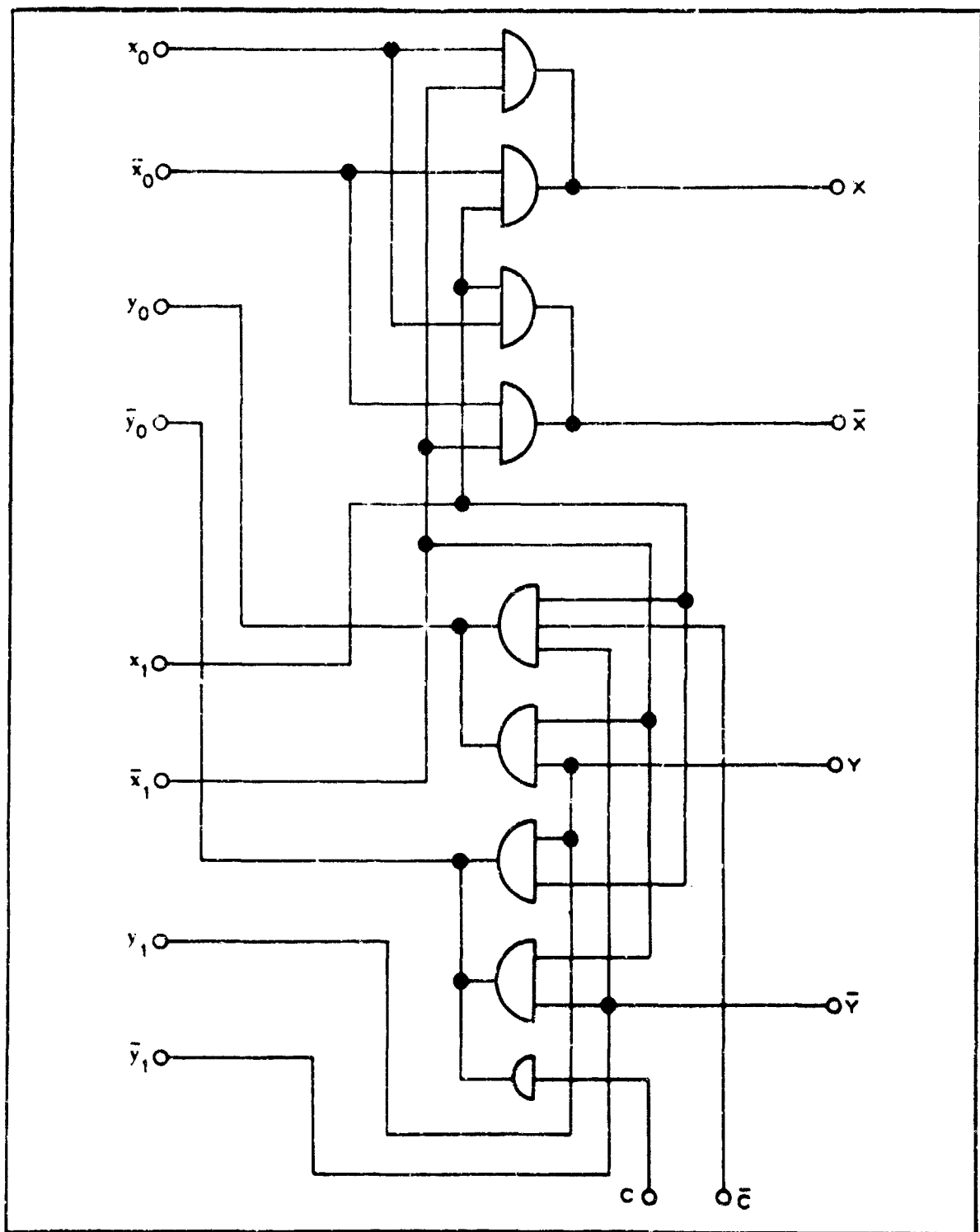
Figure VII-9 - Ring-Sum Element
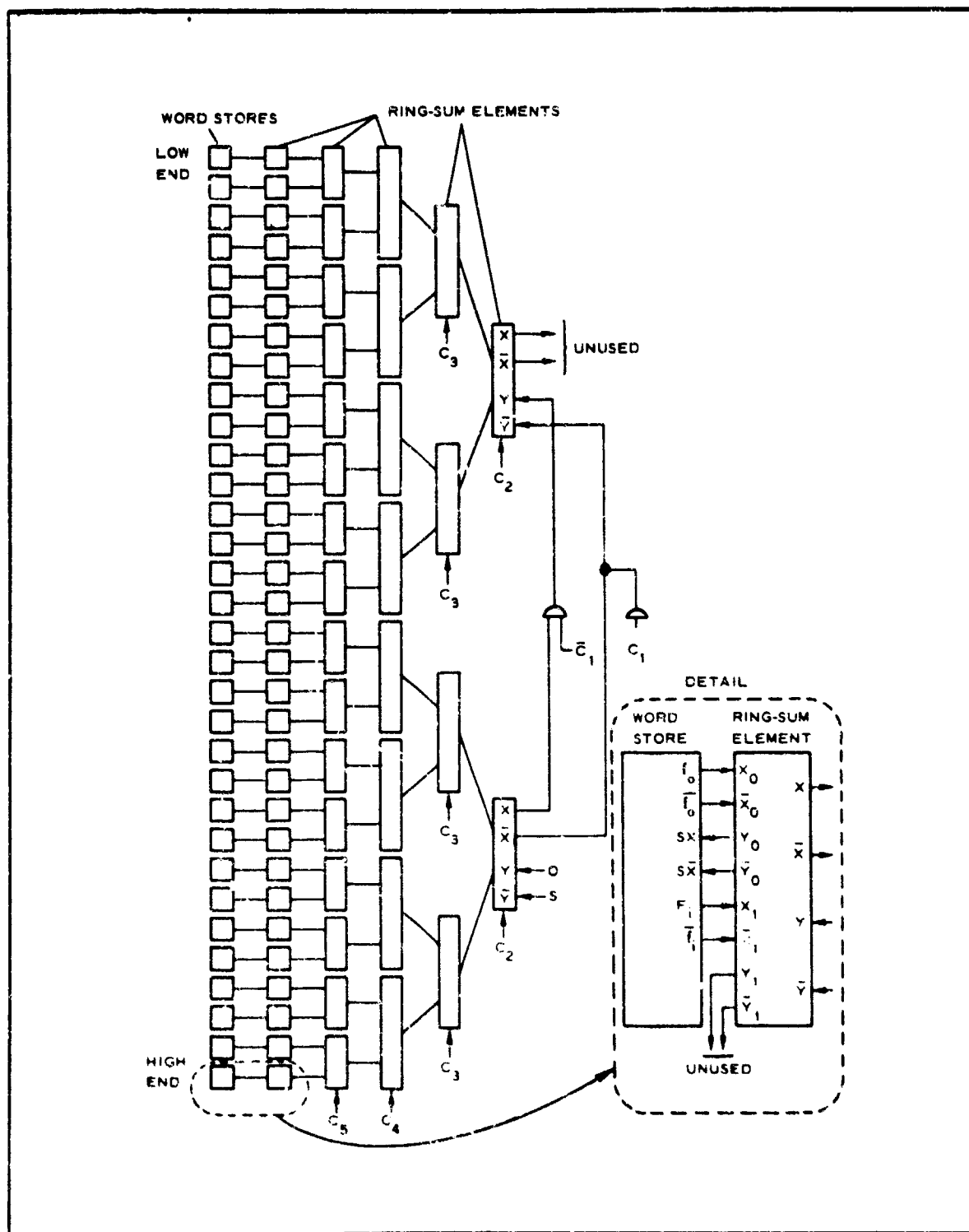
Figure VII-10 - Ring-Sum Tree for 64-Word Memory

for separating are given by: "Location i feeds location j if the n-bit representation of j is the same as that for i shifted right one place end-around." These would be the same wires as those for merging except information travels through them in the reverse direction. To simplify wiring, the same wires may be used for both phases, with the correct input and output gates being turned on to direct the information correctly (Figure VII-5). For parallel word transfer, the single wires in Figure VII-5 actually are busses.

Each word store consists of a number of digit stores plus interconnecting logic. A digit store is shown in Figure VII-6. It stores one digit of each number, $a_i$ and $b_i$, respectively, in shift register stages. The outputs $g_i = a_i \bar{b}_i$ and $t_i = a_i \vee \bar{b}_i$ are used in the comparison logic of the element. The combined input-outputs $l_i$, $e_i$, $o_i$ and $h_i$ connect to the corresponding input-outputs of other elements. These connections are shown in the following rules (the elements are numbered 0, 1, 2, . . . ., $2^{n-1} - 1$):

1. If k is even, then $e_i$ of element k connects to $l_i$ of element $1/2k$ and $o_i$ of element k connects to $l_i$ of $1/2k + 2^{n-2}$.

2. If k is odd, then $e_i$ of element k connects to $h_i$ of element $1/2(k - 1)$ and $o_i$ of element k connects to $h_i$ of element $1/2(k - 1) + 2^{n-2}$.

3. If $k < 2^{n-2}$, then $l_i$ of element k connects to $e_i$ of element 2k and $h_i$ of element k connects to $e_i$ of element 2k + 1.

4. If $k \geq 2^{n-2}$, then $l_i$ of element k connects to $o_i$ of element $2k - 2^{n-1}$ and $h_i$ of element k connects to $o_i$ of element $2k - 2^{n-1} + 1$.

The other variables in Figure VII-6 are M, which is "on" in the merging phase; S, which is "on" in the separating phase; X, which is "on" if the two words should be exchanged (for example, X = 1 if A > B in the merging phase); and $\bar{X}$, which is "on" if the words should not be exchanged.

The comparison logic and control signals M, S, M$\dot{X}$, MX, S$\dot{X}$, and SX in terconnect the digit stores of a particular element. A fast comparison circuit can be realized with a "look-ahead" circuit. This technique is similar to the "carry look-ahead" technique[a] and consists of a grouping of the logic in 2-groups or 3-groups, etc. For a comparison circuit, a p-group (for any p $\overset{\geq}{=}$ 2) is shown in Figure VII-7. It requires p "and" gates with a total of $2 + 3 + \ldots + p + p = 1/2(p^2 + 3p - 2)$ gate inputs. These groups are cascaded to form the comparison logic; Figure VII-8 shows an example of the cascade for 36-bit words. The G output of the 4-group is "on" if and only if A > B. When it is "on," it causes an exchange of A and B on the outputs (during the merging phase) by means of MX; if it is "off," M$\dot{X}$ is "on" (during merging) to cause an output with no exchange. The T outputs of the 4-group and the right-most 3-groups on each level can be eliminated as they are not used. This is also true of $t_1$ in the first digit store.

In the separating phase, the flag bits of A and B in each word store are fed into a ring-sum tree. The ring-sum tree generates the control signals S$\dot{X}$ and SX for each word store. It consists of $2^n - 2$ ring-sum elements (Figure VII-9), each of which consists of two exclusive-or circuits, each generating a true and complement output. The logical equations (for $c = 0$) are:

$$X = x_0 \oplus x_1 ,$$

$$y_0 = x_1 \oplus Y , \text{ and}$$

$$y_1 = Y .$$

When $C = 1$, $Y = 0$ and the logic is changed to force $y_0 = 0$. C is a control signal used to break the ring-sum tree into smaller pieces. An example ring-rum tree for a 64-word memory is shown in Figure VII-10.

[a] MacSorley, O. E.: "High-Speed Arithmetic in Binary Computers." Proc. IRE, Vol 49, No. 1, January 1961.

In some places on the figure, the complementary signals are not indicated; in all connections between ring-sum elements, the X and $\overline{X}$ outputs feed $x_0$ and $\overline{x}_0$ of another element, respectively, or they feed $x_1$ and $\overline{x}_1$ of another element. Similarly, the $y_0$ and $\overline{y}_0$ ($y_1$ and $\overline{y}_1$) outputs feed Y and Y of another element, respectively. During the first separating step, S is turned on and $C_1$, $C_2$, $C_3$, $C_4$, and $C_5$ are left off. In the second step, $C_1$ is turned on (S is left on); this disconnects the ring-sum tree into two parts. In the third step, $C_1$, $C_2$, and S are "on," disconnecting the tree into four parts. In the fourth step, $C_1$, $C_2$, $C_3$, and S are "on" and the tree is in eight parts. $C_1$, $C_2$, $C_3$, $C_4$, and S are "on" in the fifth step, disconnecting the tree into 16 parts. In the last step, $C_1$, $C_2$, $C_3$, $C_4$, $C_5$, and S are "on" and the tree is in 32 parts. The way words are transferred between the steps and the way control signals are turned on causes each word store to receive the correct exchange signals, SX and $S\overline{X}$ (see the separating phase discussion in Item 2 above).

On the first separating step, the longest path in the ring-sum tree for $2^n$ words goes through $2n - 1$ logic elements; on the second step, it goes through $2n - 3$ elements; on the third step it goes through $2n - 5$ elements, etc. To decrease the cycle time to a minimum, a special clock with a long interval can be used during the first separating step, a shorter interval during the second separating step, etc.

In the flagging phase, the words to be separated are flagged and the contents of words are transferred to the read requests. There may be several read requests bunched reading the same word. It would take an inordinate amount of logic to transfer the word in parallel to all such requests so in this situation only the topmost read request (the request just below the word being read) receives the data. After the separating phase, all such read requests will still be together and the topmost request can then send the data to all the others.

A control field in the low-order bits of each word identifies the word as a memory word, a read request, a read and erase request, or an erase

limit. For read requests and read and erase requests, the control field also identifies the particular output channel involved. It is desirable to arrange the control field codes so that for erase limits they are above (when read as binary numbers) those for memory words which in turn are above those for the read requests and the read and erase requests. A good control field code then is:

| $C_1$ | $C_2$ | $C_3$ | | . | . | . | . | . | $C_n$ | |
|-------|-------|-------|----|----|----|----|----|----|-------|---|
| 1 | 1 | x | x | . | . | . | x | x | | Erase limit |
| 1 | 0 | x | x | . | . | . | x | x | | Memory word |
| 0 | 1 | Channel number | | | | | | | | Read and erase request |
| 0 | 0 | Channel number | | | | | | | | Read request |

In the flagging phase, the following are to be flagged:

    1. Erase limits

    2. Read and erase requests

    3. Read requests

    4. Memory words just above read
        and erase requests.

If $C_2$ is picked for the flag, then the substitution for the flag bit of the $i^{th}$ word during flagging is:

$$C_2^{(i)} \leftarrow C_2^{(i)} \vee \bar{C}_1^{(i)} \vee \left[ C_1^{(i-1)} C_2^{(i-1)} \right]$$

$C_1$ is left alone so that it can be used to separate the requests from the erasures and erase limits after all these words have been separated from the other memory words.

The memory words can be transferred to read requests by writing the whole memory word (except its control field) into the read request, leaving the read request control field alone (the read request is just below the

memory word). Parallel transfer gates from the A word of each word store to the B word of the same word store and gates from the B word of each word store to the A word of the next lower word store are needed for this. This involves much wiring.

The flagging phase takes one time step. The exchange phase consists of one time step during which all separated words are transferred out of memory and replaced with new requests, memory words, or blank words. Checks are made to inhibit writing over any memory word.

4. CONCLUSIONS

A parallel merging-separating memory has been described. It has the advantage over a complete sorting memory of taking less time steps. Its operation is faster than a serial memory because whole words are treated at once; this time advantage is about 2 to 1. The wiring will be more complex than in a serial memory and the cost will be higher because of this and also because there are many more different kinds of elements than in a serial memory.

# APPENDIX VIII - PROBLEM SELECTION FOR A PARALLEL PROCESSOR

## 1. INTRODUCTION

This appendix presents some of the analytical results obtained in the selection of problems for implementation on a parallel processor (see Appendix VI). Parallel execution of the following mathematical methods is discussed: Jacobi's method of eigenvalue determination, relaxation solution of a system of linear algebraic equations, and numerical solution of Laplace's equation.

## 2. JACOBI'S METHOD

### a. Discussion

#### (1) General

Jacobi's method is a mathematical technique for finding the eigenvalues and eigenvectors of a real symmetric matrix. The method is based on the following well-known theorem from matrix algebra.

#### (2) Theorem 1

Let $A = (a_{ij})$ be an $n \times n$ real symmetric matrix. Then there exists an orthogonal matrix U such that

$$U'AU = D(\lambda_1, \lambda_2, \ldots, \lambda_n)$$

$$= D, \tag{1}$$

where U' denotes the transpose of U; $D = D(\lambda_1, \lambda_2, \ldots, \lambda_n)$ denotes a diagonal matrix; and $\{\lambda_i\}$ $(i = 1, 2, \ldots, n)$ are the eigenvalues of A. Since in (1) U is orthogonal,

$$AU = UD \tag{2}$$

and hence the columns of U are the eigenvectors of A.

-183-

Jacobi's method specifies the construction of a sequence of orthogonal matrices $T_1$, $T_2$, . . ., $T_k$ such that

$$T_k' T_{k-1}' \cdots T_1' A T_1 T_2 \cdots T_k = C, \tag{3}$$

where C is an $n \times n$ matrix whose off-diagonal elements are arbitrarily close to zero and whose diagonal elements are arbitrarily close to the eigenvalues of A. The columns of the matrix $T_1 T_2 \cdots T_k$ are then arbitrarily close to the eigenvectors of A.

The sequence of matrices $T_1$, $T_2$, . . ., $T_k$ is constructed as follows.

(3) Construction of $T_1$

From the elements above the main diagonal of A select the one of largest magnitude, say $a_{ij}$. Then define

$$\tan 2\theta = \frac{-a_{ij}}{\frac{1}{2}\left(a_{ii}^2 - a_{jj}^2\right)}. \tag{4}$$

Letting

and

$$\left. \begin{array}{l} c = \cos \theta \\ \\ \\ s = \sin \theta \end{array} \right\}, \tag{5}$$

$T_1$ is defined as

$$T_1 = (t_{pq}), \text{ where } t_{pq} = \begin{cases} c \text{ if } p = q = i \text{ or } p = q = j \\ s \text{ if } p = i, q = j \\ -s \text{ if } p = j, q = i \\ 1 \text{ if } p = q \neq i \text{ or } j \\ 0 \text{ otherwise} \end{cases} \tag{6}$$

More simply,

$$
T_1 = \begin{pmatrix}
1 & & & & & & & & & & \\
& 1 & & & & & & & & & \\
& & \cdot & & & & & & & & \\
& & & \cdot & & & & & & & \\
& & & & \cdot & & & & & & \\
& & & & & c & & & & s & \\
& & & & & & 1 & & & & \\
& & & & & & & 1 & & & \\
& & & & & & & & \cdot & & \\
& & & & & & & & & \cdot & \\
& & & & & & & & & & \cdot \\
& & & & & -s & & & & c & \\
& & & & & & & & & & 1 \\
& & & & & & & & & & & 1 \\
\end{pmatrix}
\begin{matrix} i \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ j \end{matrix}
\tag{7}
$$

where the unindicated terms are zeros.

If in (4) the following are set:

$$
\left.
\begin{aligned}
-a_{ij} &= \lambda \\[1em]
\frac{1}{2}\left(a_{ii}^2 - a_{jj}^2\right) &= \mu
\end{aligned}
\right\}
\tag{8}
$$

and

and $\omega$ is defined

$$
\omega = \operatorname{sgn}(\mu)\frac{\lambda}{\sqrt{\lambda^2 + \mu^2}} ,
\tag{9}
$$

$$\text{where sgn}(x) = \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ if } x = 0 \,, \\ -1 \text{ if } x \quad 0 \end{cases}$$

then one can write:

$$s = \sin \theta = \frac{\omega}{\sqrt{2\left(1 + \sqrt{1 - \omega^2}\right)}}$$

and                                                                                   (10)

$$c = \cos \theta = \sqrt{1 - \sin^2 \theta}$$

Hence, the computation of s and c involves only algebraic relationships and no computation of trigonometric functions is required.

(4) Construction of $T_{k+1}$

Assuming $T_1, T_2, \ldots T_k$ have been computed, define

$$A_k = T_k' T_{k-1}' \cdots T_1' A T_1 T_2 \cdots T_k . \tag{11}$$

Then select from the elements above the main diagonal of $A_k$, the one of largest magnitude, and calculate the elements of $T_{k+1}$ in the fashion specified by (4) through (10).

That matrices $T_i$ of the type (7) are orthogonal is easily seen by forming $T_i' T_i = I$. It is also evident that $T_{k+1}' A_k T_{k+1}$ is a real symmetric matrix if $A_k$ is, for if it is assumed that A is real and symmetric, $T_1' A T_1$ is obviously real. Further,

$$(T_1' A T_1)' = T_1' A' T_1 = T_1' A T_1 , \tag{12}$$

and hence $T_1' A T_1$ is symmetric. The general case follows by induction. It is easily seen that premultiplying a matrix $A_k$ by $T_{k+1}'$ results in a matrix $T_{k+1}' A_k$ that is identical to $A_k$ except in the $i^{th}$ and $j^{th}$ rows,

i and j being determined by the above diagonal term of $A_k$ having the largest magnitude. Similarly, postmultiplying a matrix $T'_{k+1} A_k$ by $T_{k+1}$ results in a matrix $T'_{k+1} A_k T_{k+1} = A_{k+1}$ that is identical to $T'_{k+1} A_k$ except in the $i^{th}$ and $j^{th}$ columns. A little arithmetic will show that the i, j and j, i elements of $A_{k+1}$ are zero. It may be, of course, that the "i, j" and "j, i" spots previously zeroed out in forming $A_k$ no longer will be zero in $A_{k+1}$.

However, if $t^2(A)$ is defined as the sum of the squares of the off-diagonal terms of the matrix $A_1$, it can be shown[1,a] that

$$t^2(A_{k+1}) < t^2(A_k) \qquad (13)$$

and hence the sequence A, $A_1$, $A_2$, . . . , $A_k$ generated by the Jacobi method coverges to $D = D(\lambda_1, \lambda_2, \ldots, \lambda_n)$, the diagonal matrix of the eigenvalues of A, and that the columns of $T_1 T_2 \ldots T_k$ converge to the eigenvectors of A.

b. Parallel Execution

The method of Jacobi, as outlined above, lends itself well to parallel computation. Matrix operations are, of course, well suited for parallel computation. As an example, consider the product $C = AB$ of two matrices $A = (a_{ij})$, $B = (b_{ij})$. Now in C, the i, j element is

$$c_{ij} = \sum_{k=1}^{n} c_{ik} b_{kj} . \qquad (14)$$

That is, the element in the i, j spot of C is just the dot product of the $i^{th}$ row of A and the $j^{th}$ column of B. Clearly, given A and B, each of the elements of C may be calculated independently of the others. And for each element of C the multiplications involved in the corresponding dot product

---

[a] Superior numbers in the text refer to references under Subhead 7 on Page 220.

may be done in parallel and the summing involved may be treed. The extensive matrix operations involved in the Jacobi method are then well suited to parallel computation.

There are two computational aspects of the Jacobi method for which capabilities resident in parallel processors having sorting memories are ideally suited (see Appendix VI). These aspects are (1) the determination for a matrix A of the above diagonal element of largest magnitude, and (2) the test for convergence, namely,

$$t^2(A_i) < \epsilon \tag{15}$$

for some given epsilon.

The test (15) for convergence may be replaced by requiring that the magnitude of the largest off-diagonal element of $A_i$ be less than some given epsilon.

Since each of the two computational aspects cited above involves the determination of the largest member of a given set of elements, the rapid sort capability of Machines I or II (Appendixes VI and XV) may be profitably brought to bear in their execution.

## 3. THE RELAXATION TECHNIQUE

### a. Discussion

Relaxation is a term originally applied by R. V. Southwell to a class of iterative methods for solving a system of linear equations. The term has since come to connote a broad class of methods for the approximate reformulation of physical problems in terms of systems of linear equations to be solved. An example of this expanded use of the term relaxation is offered under Item 4 below where a numerical solution to Laplace's equation is discussed. In the strict sense, the relaxation technique provides a method for solving a system of linear algebraic equations expressed in matrix form as

$$AX = B, \tag{16}$$

where A is an $n \times n$ coefficient matrix of known constants, $X = (x_1, x_2, x_3, \ldots, x_n)$ is a column vector of unknowns, and $B = (b_1, b_2, \ldots, b_n)$ is a column vector of known constants.

The relaxation technique is an iterative procedure that specifies a sequence $X_1, X_2, \ldots, X_k$ where $X_i = (x_1^i, x_2^i, \ldots, x_n^i)$ of approximations that converge to the solution vector X. Discussions of necessary and sufficient conditions for convergence may be found in references 1 through 4. The technique assumes an initial guess, $X_1$, and computes successively vectors $R_i = (r_1^i, r_2^i, \ldots, r_n^i)$ of "residuals" defined as

$$R_i = B - AX_i \tag{17}$$

for $i = 1, 2, \ldots, k$.

The residual vector $R_i$ provides a measure of the closeness of the approximation $X_i$ to X. Based on a residual vector $R_i$, the relaxation technique specifies a new approximation $X_{i+1}$. The process continues until the elements of the residual vector are sufficiently close to zero to satisfy a pre-established convergence criterion such as $R_i \cdot R_i < \epsilon$ or

$$\max_k \left[ \left| r_k^i \right| \right] < \epsilon .$$

Given a residual vector $R_i = (r_1^i, r_2^i, \ldots, r_n^i)$, the relaxation procedure specifies a new approximation $X_{i+1}$ of the form

$$X_{i+1} = X_i + \lambda_p U_p . \tag{18}$$

where $U_p$ is the $p^{th}$ coordinate vector, namely

$$U_p = (\delta_{1p}, \delta_{2p}, \ldots \delta_{np}) ; \tag{19}$$

and $\lambda_p$ is a constant to be chosen such that the $p^{th}$ element, $r_p^{i+1}$, of the residual vector $R_{i+1} = B - A X_{i+1}$, is zero. P may be specified in a

cyclic order, for example in terms any permutation of the integers 1, 2, . . ., n (n being the order of the matrix A), or according to some predetermined criterion. The process of choosing $X_{i+1} = X_i + \lambda_p U_p$ in terms of a cyclic determination of p is known as the Gauss-Seidel iteration.[5]

More rapid convergence of the relaxation technique is obtained if $\lambda_p$ is chosen so that

$$\left[ |\lambda_p| \right] = \overset{max}{\underset{i}{}} \left[ |\lambda_i| \right] \tag{20}$$

rather than specifying p in a cyclic fashion.

It is possible to determine $\lambda_p$, where p = 1, 2, . . ., n, as follows: For a given p and present approximation $X_i = (x_1^i, x_2^i, \ldots, x_n^i)$, the requirement that $r_p^{i+1} = 0$ means that if the residual vector $R_{i+1} = (r_1^{i+1}, r_2^{i+1}, r_n^{i+1})$ then the dot product $R_{i+1} \cdot U_p = 0$.

Now $R_{i+1} = B - AX_{i+1}$ and $X_{i+1} = X_i + \lambda_p U_p$.

Hence $(B - AX_{i+1}) \cdot U_p = 0$ and

$$\lambda_p = \frac{b_p - \displaystyle\sum_{k=1}^{n} a_{pk} x_k^i}{a_{pp}}. \tag{21}$$

Letting

$$\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n), \tag{22}$$

then

$$\lambda = \left( r_1^i / a_{11}, \ r_2^i / a_{22}, \ \ldots, \ r_n^i / a_{nn} \right). \tag{23}$$

At each stage of the relaxation iteration, a new approximation to the solution vector X is specified in terms of the last approximation and the element of (23) having maximum magnitude.

The rate of convergence of the relaxation process may be increased by modifying the value of $\lambda_p$ used in the iteration. If $\lambda_p$ is replaced by

$$\lambda_p^* = w\lambda_p \, , \tag{24}$$

it is known that for

$$0 < w < 2 \tag{25}$$

convergence of a relaxation iteration is preserved. The factor w is called an acceleration parameter or relaxation factor. The term under [over] relaxation is applied · the case where $0 < w < 1$ $[1 < w < 2]$. (It must be stressed here that the acceleration parameter w is used to accelerate, not establish, convergence of a relaxation iteration.) The central problem associated with the use of an acceleration parameter w is to determine the optimal value, $w_{opt}$, for w; that is, the value of w for which the convergence rate of the relaxation iteration is maximized. The theoretical determination of $w_{opt}$ for the relaxation solution of a system of equations expressed in matrix form as

$$AX = B \tag{26}$$

proceeds as follows. Let the matrix A be represented as

$$A = \left( {}_E D^F \right) \tag{27}$$

where E, D, and F are lower triangular, diagonal, and upper triangular matrices. Defining a matrix H as

$$H = -(D + E)^{-1}F \tag{29}$$

and denoting by S(H) the spectrum of H, then compute

$$\mu = \max \left[ S(H) \right] . \tag{30}$$

That is, $\mu$ is the eigenvalue of H having the largest magnitude. If the relaxation iteration is convergent for the system (26), then $|\mu| < 1$ (ref 2) and $w_{opt}$ may be computed as

$$w_{opt} = \frac{2}{1 + (1 - \mu)^{1/2}} .$$ 
$$(31)$$

Observe that implementation of the derivation of $w_{opt}$ presented above involves the solution of an eigenvalue problem that may be at least as difficult as the original problem. Forsythe[3] makes the discouraging observation that no generally acceptable technique for accurately approximating $w_{opt}$ as the relaxation iteration proceeds is known. Householder[6] recently confirmed this observation.

### b.   Parallel Execution

The relaxation method outlined above involves the repeated execution of the operations of matrix multiplication and addition, multiplication of a vector by a scalar, and searching a set for the element of largest magnitude. As was pointed out under Item 2, b above, these operations are well suited to parallel execution, and the operation of finding in a set the element of largest magnitude may be rapidly accomplished on a parallel processor having sorting capability.

## 4.   NUMERICAL SOLUTION TO LAPLACE'S EQUATION

### a.   Discussion

The numerical solution of Laplace's equation over a rectangular region R with boundary $\Gamma R$ is discussed here. Assume that R is partitioned by an equally spaced rectangular mesh and that Dirichlet boundary conditions are specified. Given a function $u(x, y)$ for which Laplace's equation obtains over R, write

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 .$$ 
$$(32)$$

Letting the interval for the mesh over R be denoted by $\Delta$, the partial derivatives for $u(x, y)$ may be approximated by

$$\frac{\partial u}{\partial x} = \frac{u(x + \Delta, y) - u(x, y)}{\Delta}$$

$$\frac{\partial u}{\partial y} = \frac{u(x, y + \Delta) - u(x, y)}{\Delta}$$

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x + \Delta, y) - 2u(x, y) + u(x - \Delta, y)}{\Delta^2}$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u(x, y + \Delta) - 2u(x, y) + u(x, y - \Delta)}{\Delta^2}$$

(33)

and then the difference equation counterpart of (32) may be written as

$$u(x, y) = \frac{1}{4}\left[u(x + \Delta, y) + u(x - \Delta, y) + u(x, y + \Delta) + u(x, y - \Delta)\right], \quad (34)$$

Equation (34) approximates $u(x, y)$ at each interior mesh point of R by the average of "north, south, east, west neighbors." Other difference equation approximations to $u(x, y)$ at interior points of R are

$$u(x, y) = \frac{1}{4}\left[u(x + \Delta, y + \Delta) + u(x + \Delta, y - \Delta) + u(x - \Delta, y + \Delta) + \right.$$

$$\left. u(x - \Delta, y - \Delta)\right] \quad (35)$$

and

$$u(x, y) = \frac{1}{5}\left[u(x + \Delta, y) + u(x - \Delta, y) + u(x, y + \Delta) + u(x, y - \Delta)\right] +$$

$$\frac{1}{20}\left[u(x + \Delta, y + \Delta) + u(x + \Delta, y - \Delta) + u(x - \Delta, y + \Delta) + u(x - \Delta, y - \Delta)\right].$$

(36)

Approximations (34) and (35) are often referred to as "five-point" formulas and (36) as a "nine-point" formula. It is easily seen that approximations (34), (35), and (36) represent $u(x, y)$ in terms of $+$, $\times$, and $\square$ patterns of neighbors, respectively, and they are referred to here as approximations A, B, and C.

An easily established relation between approximations A, B, and C [(34), (35), and (36)] is given by

$$C = \frac{2}{3}A + \frac{1}{3}B . \tag{37}$$

Iterative solutions to Laplace's equation based on approximations A, B, or C converge[3] and often are called "relaxation solutions." A sequential iterative solution would proceed by ordering the interior mesh points of a region R and cyclicly applying one of the approximations A, B, or C over the ordering until some specified convergence criterion is met. In a sequential pass over the ordered interior mesh points of R, two possibilities for updating the values for u(x, y) at each interior mesh point are available: (1) as each new approximation to u(x, y) is generated at a point, it is made available for subsequent calculations in the pass, and (2) each pointwise approximation to u(x, y) made in a given pass uses only point values available at the end of the preceding pass. The former [latter] method of updating often is called the method of successive [simultaneous] displacements.

If the interior mesh points are ordered, say as $p_1$, $p_2$, . . . ., $p_k$, and at each point $p_i$ the value of u(x, y) is regarded as a variable $x_i$ to be determined, then each of the methods A, B, or C of approximating u(x, y) over R may be written in matrix form as

$$P X = Q \tag{38}$$

where $X = (x_1, x_2, . . . ., x_k)$ is a vector of unknown corresponding to the values of u(x, y) at the interior mesh points of R, P is a coefficient matrix of known constants determined by the type of approximation (A, B, or C) being used, and Q is a vector of known constants determined by the approximation being used and known boundary values for u(x, y). The system (38) may be solved by relaxation methods discussed under Item 3 above.

If for a given method of approximation to u(x, y) over the interior of R, the corresponding matrix [as cited in (38)] is constructed and $w_{opt}$ (see

Item 3) is calculated, then $w_{opt}$ may be used to increase the rate of convergence for the method of successive displacements described above.

For approximation A, the procedure would be as follows. For the function $u(x, y)$, a residual $r(x, y)$ is defined at each interior mesh point of R as

$$r(x, y) = u(x + \Delta, y) + u(x - \Delta, y) + u(x, y + \Delta) + u(x, y - \Delta) - 4u(x, y) .$$

(39)

Then specify a new approximation, say $u'(x, y)$, as

$$u'(x, y) = u(x, y) + \frac{w_{opt}}{4} r(x, y) .$$

(40)

Note the correspondence of (40) and (18). Similar modifications of approximations B and C are readily specified. Although modification of the method of successive displacements by the use of $w_{opt}$ in the fashion of (40) increases the convergence rate, the use of $w_{opt}$ in conjunction with the method of simultaneous displacements is of no profit. [3]

The numerical solution to Laplace's equation over a rectangular region partitioned by an equally spaced rectangular mesh is specified easily in terms of approximations A, B, or C and the methods of simultaneous or successive displacements. An immediate question arises as to which of the available techniques offers the most rapid convergence. To compare the relative merits of the techniques outlined above, code ITEST was written in FORTRAN IV for the IBM 1410. ITEST will solve Laplace's equation over a 9-by-9 square mesh of equal mesh spacings using approximations A, B, or C (or combinations) in conjunction with simultaneous or successive displacements. Table VIII-1 lists some results obtained using ITEST. For each of the three runs listed, $u(x, y)$ was specified to be zero on the boundary. The true solution for $u(x, y)$ was then $u(x, y) = 0$ in all cases. In run 1, $u(x, y)$ initially was specified to be zero at each interior mesh point except at the "center" point, which was specified as 1.0. In runs 2 and 3, $u(x, y)$ was specified as 1.0 at

TABLE VIII-1 - RESIDUES AFTER TWELVE

ITERATIONS FOR RUNS 1, 2, AND 3

| Approximation sequence | Residues after 12 iterations | | |
|---|---|---|---|
| | Run 1 | Run 2 | Run 3 |
| A, A, A, . . . | 0.606 | 15.482 | 6.458 |
| B, B, B, . . . | 0.218 | 5.989 | 1.071 |
| C, C, C, . . . | 0.506 | 12.856 | 4.567 |
| A, B, A, B, . . . | 0.380 | 9.602 | 2.746 |
| A, C, A, C, . . . | 0.554 | 14.105 | 5.436 |
| B, C, B, C, . . . | 0.346 | 8.757 | 2.285 |
| A, B, C, A, B, C, . . . | 0.418 | 10.581 | 3.267 |

each interior mesh point. For each of the three runs, each of seven different combinations of approximations A, B, and C was used for 12 iterative passes over the mesh. The seven combinations of A, B, and C are listed in column 1 of Table VIII-1. In runs 1 and 2, the method of simultaneous displacements was used while run 3 employed successive displacements.

For each iterative pass over the mesh, a "residue" term was calculated. The residue term is just the sum of the absolute value of the errors in the approximation to u(x, y) at the interior mesh points. Columns 2, 3, and 4 of Table VIII-1 list the residue term computed after the twelfth iterative pass for each of the seven combinations of A, B, and C for runs 1, 2, and 3, respectively. Figures VIII-1 through VIII-9 contain the pointwise approximations to u(x, y) obtained after 12 iterative passes over the 9-by-9 mesh on runs 1, 2, and 3 using successive approximations A, A, A, . . . ., B, B, B, . . . ., and C, C, C, . . . .

Inspection of the table and figures cited above reveals that for the methods tested, the most rapid convergence is obtained by using the method of successive displacements and approximation B, (X). The convergence rate

o   o   o   o   o   o   o   o   o
●   ●   ●   ●   ●   ●   ●   ●   ●

o   0.006   0.0*   0.016   0.0   0.016   0.0   0.006   o
●   ●       ●      ●       ●     ●       ●     ●        ●

c   0.0   0.023   0.0   0.034   0 0   0.023   0.0   o
●   ●     ●       ●     ●       ●     ●       ●      ●

o   0.016   0.0   0.042   0.0   0.042   0.0   0.036   o
●   ●       ●     ●       ●     ●       ●     ●        ●

o   0.0   0.034   0.0   0.050   0.0   0.034   0.0   o
●   ●     ●       ●     ●       ●     ●       ●      ●

o   0.016   0.0   0.042   0.0   0.042   0.0   0.016   o
●   ●       ●     ●       ●     ●       ●     ●        ●

o   0.0   0.023   0 0   0.034   0.0   0.023   0.0   o
●   ●     ●       ●     ●       ●     ●       ●      ●

o   0.006   0.0   0.016   3.0   0.016   0.3   0.006   o
●   ●       ●     ●       ●     ●       ●     ●        ●

o   o   o   o   o   o   o   o   o
●   ●   ●   ●   ●   ●   ●   ●   ●

ITERATION   1   2   3   4   5   6   7   8   9   10   11   12

RESIDUE   1.0   1.0   1.0   0.9844   0.9531   0.9102   0.8584   0.8081   0.7534   0.7032   0.6516   0.6060

*DUE TO THE APPROXIMATING FORMULA A (+) AND THE INITIAL APPROXIMATION FOR u(x, y), VIZ:

$$u(i, j) = \begin{cases} 1.0 \text{ FOR } (i, j) = (5, 5) \\ 0 \text{ OTHERWISE} \end{cases}$$

THE ITERATIVE APPROXIMATIONS FOR u(x, y) AT THE INTERIOR POINTS ALTERNATE BETWEEN ZERO AND NONZERO VALUES. THE VALUES FOR u(2, 2) u(2, 3), AND u(2, 4) AFTER ITERATION 11 WERE 0.0 0.0131 AND 0.0, RESPECTIVELY.

Figure VIII-1 - Run 1, Simultaneous Displacements, Approximation A (+), 12 Iterations
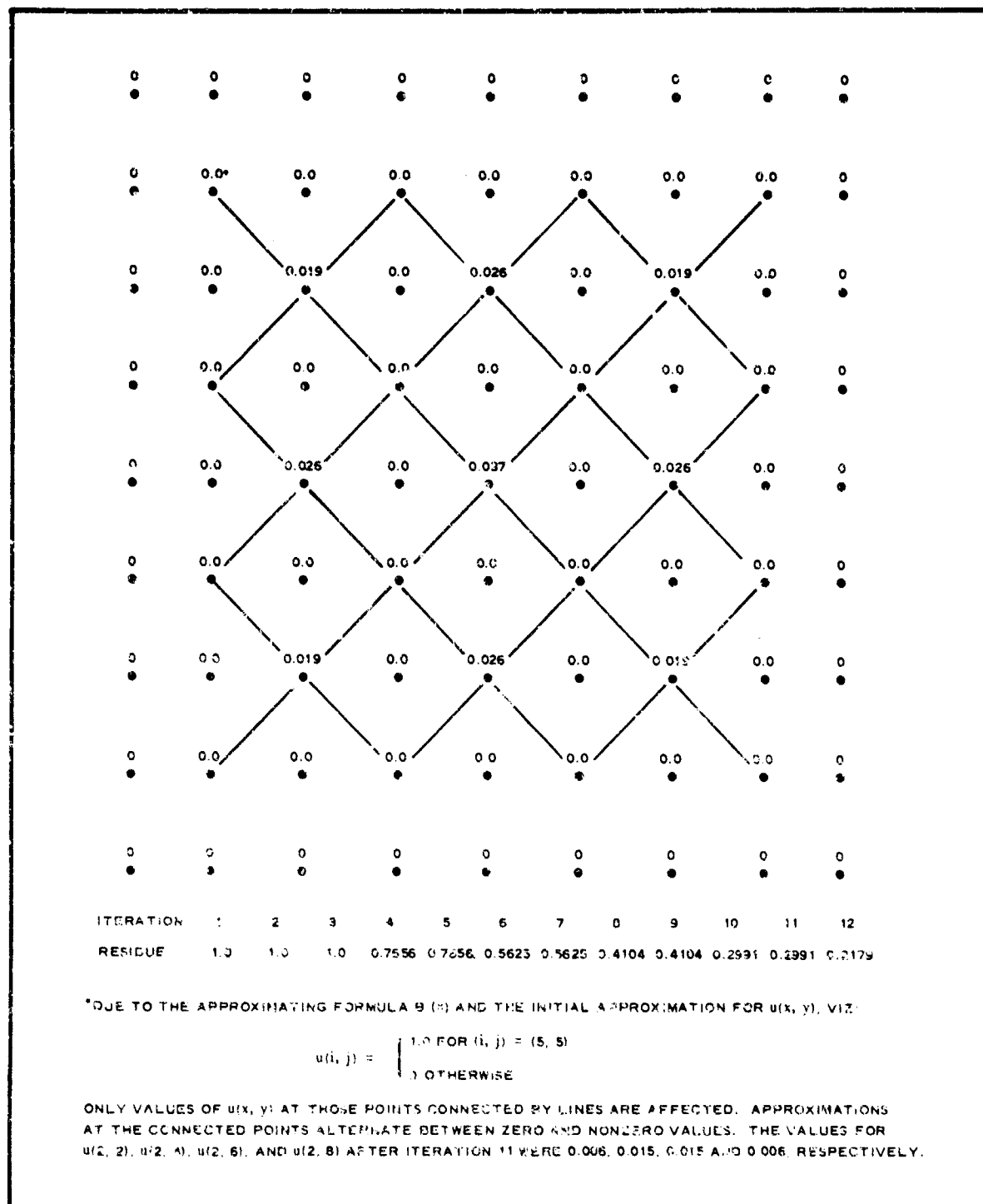
ITERATION : 2 3 4 5 6 7 8 9 10 11 12

RESIDUE 1.0 1.0 1.0 0.7556 0.7856 0.5623 0.5625 0.4104 0.4104 0.2991 0.2991 0.2179

*DUE TO THE APPROXIMATING FORMULA 9 (5) AND THE INITIAL APPROXIMATION FOR u(x, y), VIZ:

$$u(i, j) = \begin{cases} 1.0 \text{ FOR } (i, j) = (5, 5) \\ 0 \text{ OTHERWISE} \end{cases}$$

ONLY VALUES OF u(x, y) AT THOSE POINTS CONNECTED BY LINES ARE AFFECTED. APPROXIMATIONS
AT THE CONNECTED POINTS ALTERNATE BETWEEN ZERO AND NONZERO VALUES. THE VALUES FOR
u(2, 2), u(2, 4), u(2, 6), AND u(2, 8) AFTER ITERATION 11 WERE 0.006, 0.015, 0.015 AND 0.006, RESPECTIVELY.

Figure VIII-2 - Run ... ltaneous Displacements, Approximation B (X),
11 . s
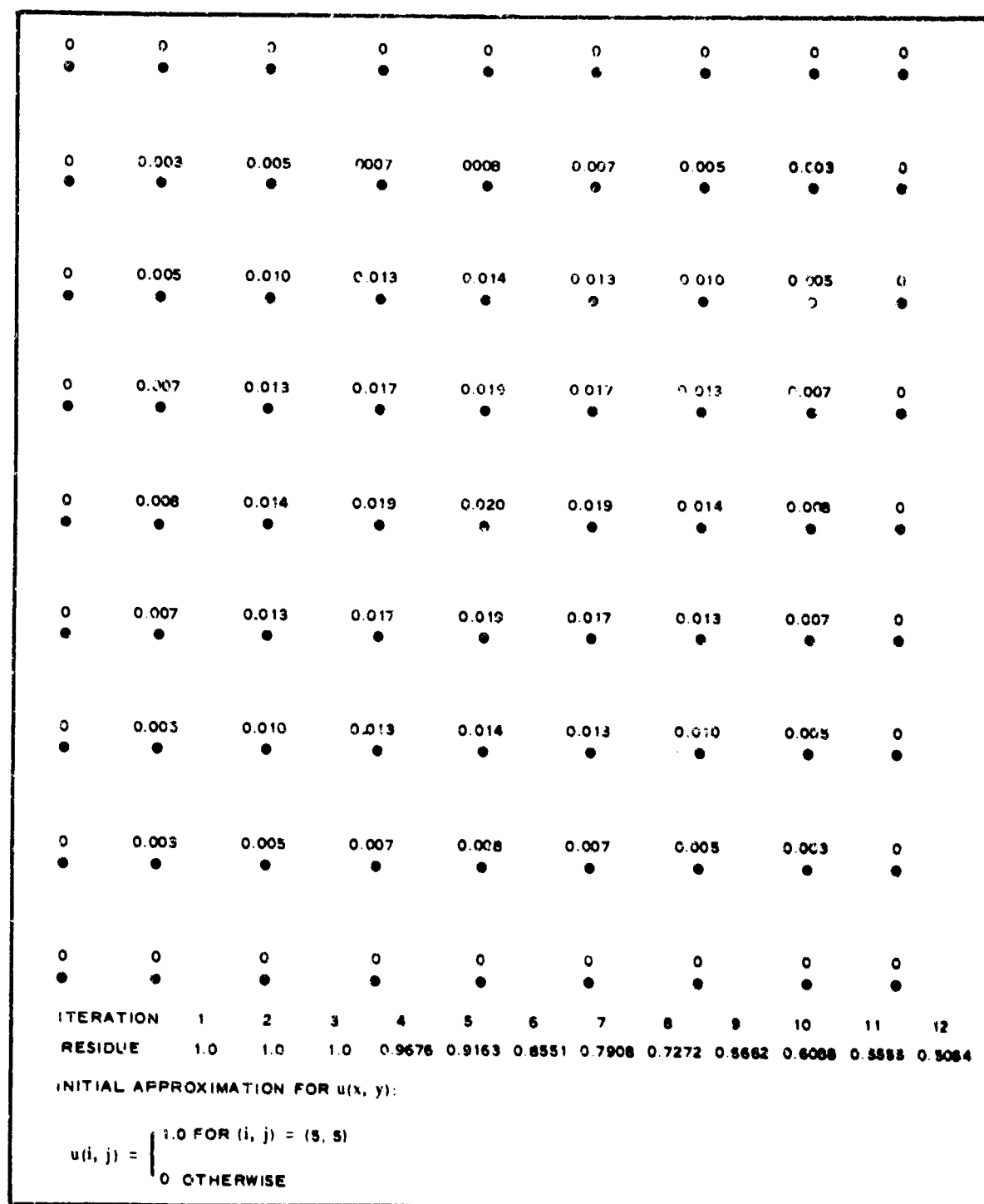
Figure VIII-3 - Run 1, Simultaneous Displacements, Approximation C (☐).
12 Iterations
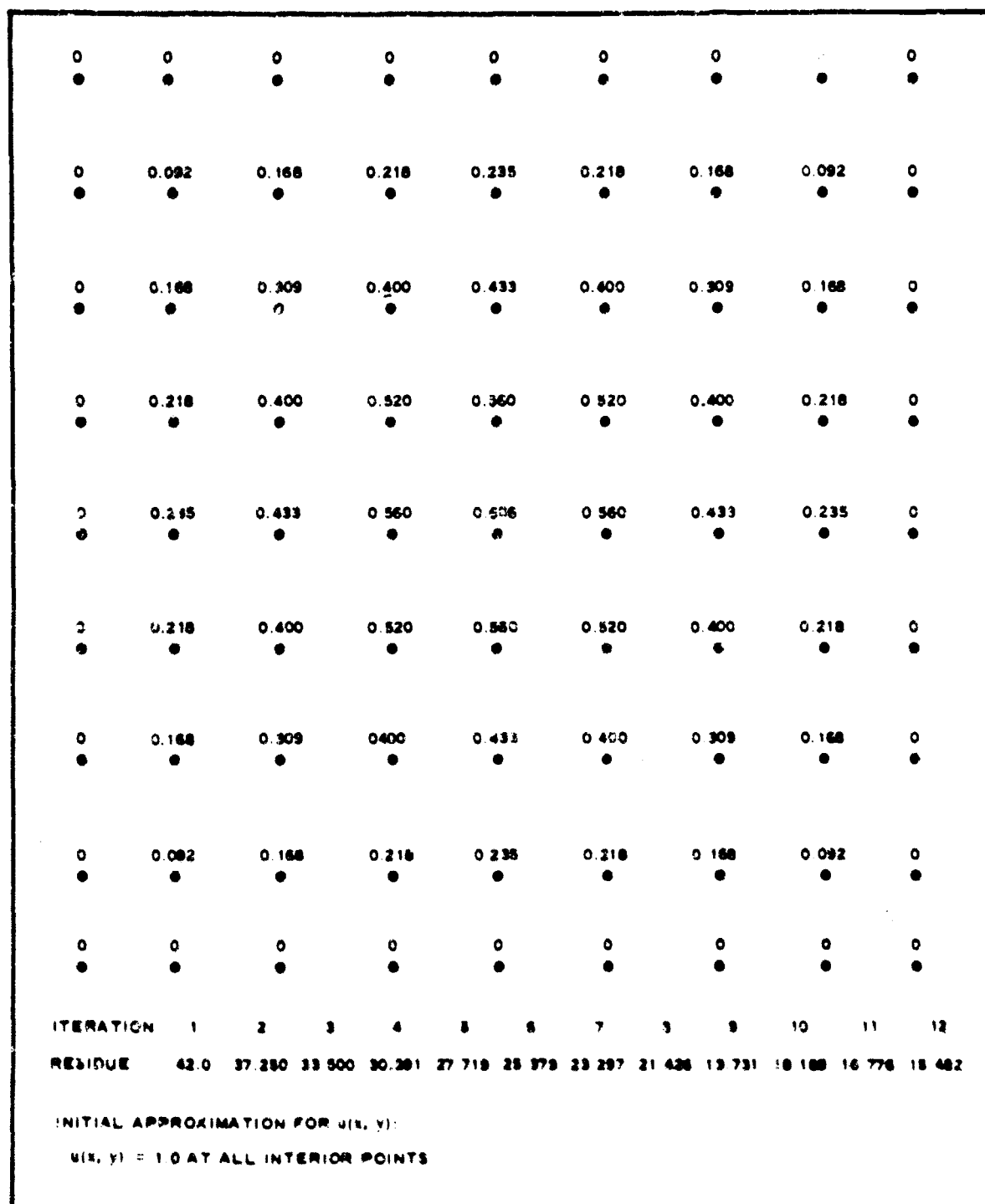
Figure VIII-4 - Run 2, Simultaneous Displacements. Approximation A (•). 12 Iterations

Figure VIII-5 - Run 2. Simultaneous Displacements. Approximation B (X).
12 Iterations

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0.075 | 0.138 | 0.180 | 0.194 | 0.180 | 0.138 | 0.075 | 0 |
| 0 | 0.138 | 0.255 | 0.332 | 0.359 | 0.332 | 0.255 | 0.138 | 0 |
| 0 | 0.180 | 0.332 | 0.433 | 0.468 | 0.433 | 0.332 | 0.180 | 0 |
| 0 | 0.195 | 0.359 | 0.468 | 0.506 | 0.468 | 0.359 | 0.195 | 0 |
| 0 | 0.180 | 0.332 | 0.433 | 0.408 | 0.413 | 0.332 | 0.180 | 0 |
| 0 | 0.138 | 0.255 | 0.332 | 0.359 | 0.332 | 0.255 | 0.138 | 0 |
| 0 | 0.075 | 0.138 | 0.180 | 0.194 | 0.180 | 0.138 | 0.075 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| ITERATION | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RESIDUE | 40.800 | 35.610 | 31.560 | 28.237 | 25.410 | 22.953 | 20.785 | 18.852 | 17.118 | 15.554 | 14.139 | 12.856 |

INITIAL APPROXIMATION FOR u(x, y):

u(x, y) = 1.0 AT ALL INTERIOR POINTS

Figure VIII-6 - Run 2, Simultaneous Displacements, Approximation C (□), 12 Iterations

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|
| • | • | • | • | • | • | • | • | • |

| 0 | 0.052 | 0.092 | 0.114 | 0.118 | 0.102 | 0.074 | 0.037 | 0 |
|---|---|---|---|---|---|---|---|---|
| • | • | • | • | • | • | • | • | • |

| 0 | 0.092 | 0.161 | 0.199 | 0.204 | 0.178 | 0.127 | 0.064 | 0 |
|---|---|---|---|---|---|---|---|---|
| • | • | • | • | • | • | • | • | • |

| 0 | 0.114 | 0.199 | 0.246 | 0.250 | 0.217 | 0.155 | 0.078 | 0 |
|---|---|---|---|---|---|---|---|---|
| • | • | • | • | • | • | • | • | • |

| 0 | 0.118 | 0.204 | 0.250 | 0.255 | 0.220 | 0.157 | 0.079 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0 | • | • | • | • | • | • | • | • |

| 0 | 0.103 | 0.178 | 0.217 | 0.220 | 0.190 | 0.135 | 0.068 | 0 |
|---|---|---|---|---|---|---|---|---|
| • | • | • | • | • | • | • | • | • |

| 0 | 0.074 | 0.127 | 0.155 | 0.157 | 0.135 | 0.096 | 0.046 | 0 |
|---|---|---|---|---|---|---|---|---|
| • | • | • | • | • | • | • | • | • |

| 0 | 0.037 | 0.064 | 0.078 | 0.079 | 0.065 | 0.048 | 0.024 | 0 |
|---|---|---|---|---|---|---|---|---|
| • | • | • | • | • | • | • | • | • |

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|
| • | • | • | • | • | • | • | • | • |

| ITERATION | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RESIDUE | 38.057 | 31.138 | 26.082 | 22.115 | 18.879 | 16.117 | 13.867 | 11.927 | 10.242 | 8.789 | 7.537 | 6.458 |

INITIAL APPROXIMATION FOR u(x, y):

u(x, y) = 1.0 AT ALL INTERIOR POINTS

Figure VIII-7 - Run 3, Successive Displacements, Approximation A (+),
12 Iterations

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| • | • | • | • | • | • | • | • | • |
| 0 | 0.010 | 0.018 | 0.023 | 0.025 | 0.023 | 0.018 | 0.010 | 0 |
| • | • | • | • | • | • | • | • | • |
| 0 | 0.015 | 0.028 | 0.036 | 0.040 | 0.036 | 0.028 | 0.015 | 0 |
| • | • | • | • | • | • | • | • | • |
| 0 | 0.017 | 0.031 | 0.041 | 0.044 | 0.041 | 0.031 | 0.017 | 0 |
| • | • | • | • | • | • | • | • | • |
| 0 | 0.016 | 0.029 | 0.038 | 0.041 | 0.038 | 0.029 | 0.016 | 0 |
| • | • | • | • | • | • | • | • | • |
| 0 | 0.012 | 0.023 | 0.030 | 0.032 | 0.030 | 0.023 | 0.012 | 0 |
| • | • | • | • | • | • | • | • | • |
| 0 | 0.008 | 0.015 | 0.019 | 0.021 | 0.019 | 0.015 | 0.008 | 0 |
| • | • | • | • | • | • | • | • | • |
| 0 | 0.004 | 0.007 | 0.009 | 0.010 | 0.010 | 0.007 | 0.004 | 0 |
| • | • | • | • | • | • | • | • | • |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| • | • | • | • | • | • | • | • | • |

ITERATION   1   2   3   4   5   6   7   8   9   10   11   12

RESIDUE   31.081  22.520  16.682  12.471  9.323  6.939  5.124  3.763  2.754  2.012  1.468  1.071

INITIAL APPROXIMATION FOR u(x, y):

u(x, y) = 1.0 AT ALL INTERIOR POINTS

Figure VIII-8 - Run 3, Successive Displacements, Approximation B (X),
12 Iterations

-204-

Figure VIII-9 - Run 3, Successive Displacements, Approximation C (□ ), 12 iterations

-205-

for B could have been accelerated by the use of $w_{opt}$. It will be noted that in solving Laplace's equation over a mesh by methods A, B, or C, the sum of errors in the approximation of $u(x, y)$ at interior points will remain a constant until the iterative procedure successively spreads the error at a point(s) to the boundary; it is only when boundary values are brought to bear that the total pointwise error in the interior of the mesh can be reduced. The method of simultaneous displacements could be implemented easily on the parallel processor described in Appendix VI. However, the slow rate of convergence obtained using simultaneous displacements and the difficulty of obtaining applicable acceleration parameters make the method somewhat unattractive, even for parallel processors.

### b. Mesh Fill In

Iterative numerical solutions to Laplace's equation over a mesh begin with the assumption of some initial values for $u(x, y)$ at interior points. Clearly, the greater the accuracy of the initial approximations, the more rapid should be the convergence. A method is described here for filling the interior of a mesh rapidly with accurate initial approximations to $u(x, y)$ based on known boundary values. This will be confined to the 9-by-9 grid previously cited. Extension of the method to any $(2^n + 1)$ by $(2^n + 1)$ mesh is immediate.

Let the value of $u(x, y)$ be denoted at points of the 9-by-9 mesh as $u(i, j)$, with i and j being determined as in matrix notation. Since $\{u(5, 1), u(5, 9), u(1, 5), u(9, 5)\}$ are known, $u(5, 5)$ may be approximated by A. Knowing $u(5, 5)$, $\{u(3, 3), u(3, 7), u(7, 3), u(7, 7)\}$ may be approximated by B, then $\{u(5, 3), u(5, 7), u(3, 5), u(3, 7)\}$ by A, etc. For a 9-by-9 mesh, five such passes are required for complete fill-in of interior points. Figure VIII-10 illustrates how the fill-in proceeds for each pass. The numbers above the points in the mesh of this illustration indicate the pass in which corresponding approximations to $u(x, y)$ were made. The approximations made during each pass depend only on boundary values or results of the previous passes, or both; hence, they are amenable to parallel

-206-
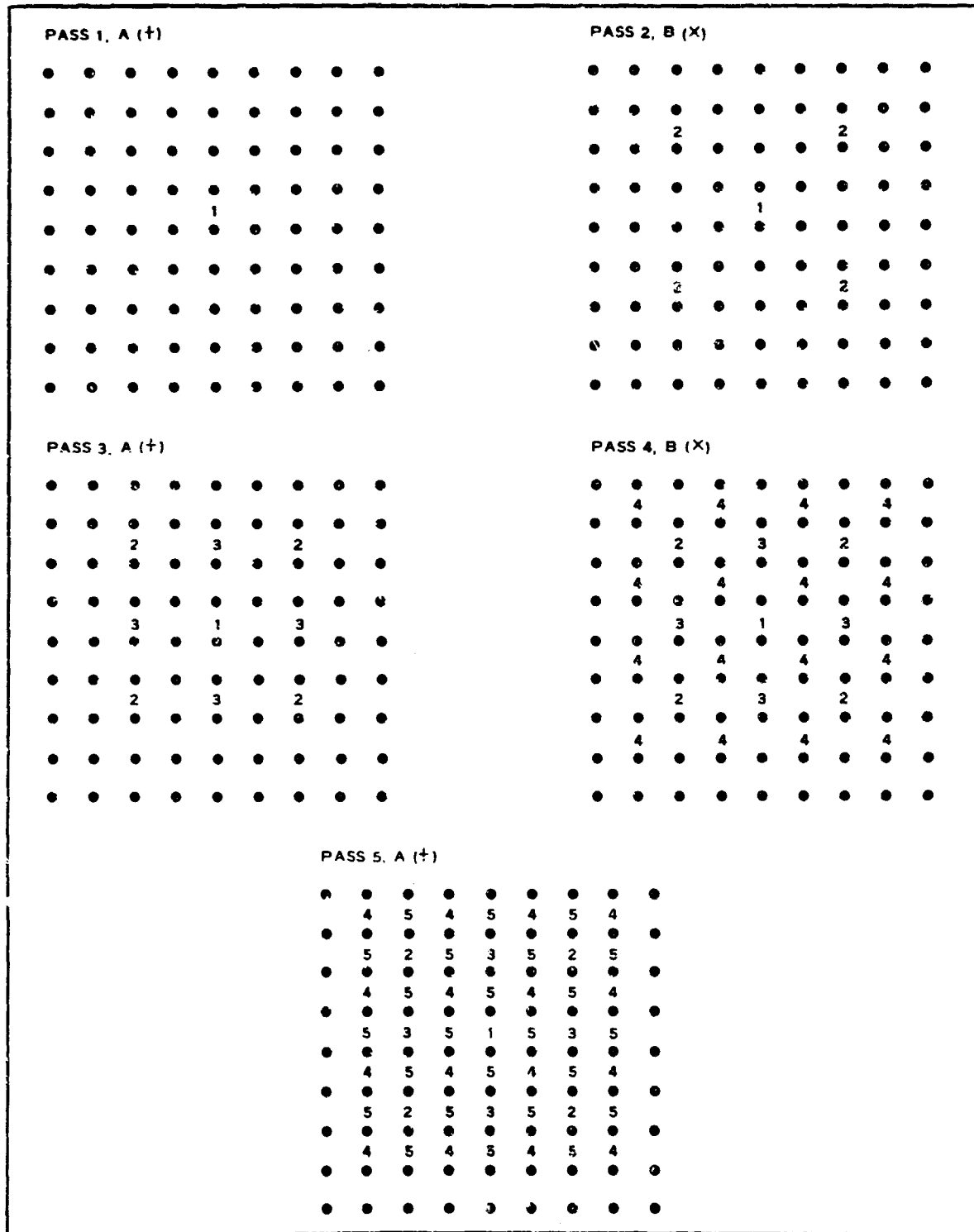
Figure VIII-10 - Parallel Fill-In

computation. Accordingly, the above method of mesh fill-in is referred to as parallel fill-in (PFI). PFI can be executed readily on parallel processors where rapid, universal communication between processing elements is available.

To check the accuracy of PFI, code FILIN was written in FORTRAN IV for the IBM 1410. Given values for u(x, y) on the boundary of a 9-by-9 rectangular mesh of equal spacing, FILIN calculates initial approximations on u(x, y) at interior points by PFI. FILIN was run for 11 different sets of boundary conditions. Figures VIII-11 through VIII-21 give the boundary conditions and resultant fill-in based on PFI for the 11 runs. Inspection of these figures reveals the excellent results achieved by PFI for the boundary conditions specified. Although the implementation of PFI is more suitable to parallel processors, its accuracy is such that it is to be recommended for use on sequential machines.

c.  Parallel Execution

The numerical solution to Laplace's equation over a mesh by simultaneous displacements is structurally well suited to parallel computation. For a parallel processor of sufficient size, a processing unit could be assigned to each interior mesh point. Each unit then would compute and store, in an iterative fashion, approximations to u(x, y) at its assigned point. The communication capabilities of Machines I or II (Appendixes VI and XV) would allow the use of any combination of the approximations A, B, or C. In the event that the number of interior mesh points exceeded the number of processing units, each unit could be assigned a block of interior mesh points and the iteration could proceed "parallel by block and sequential by point within a block. " However, although the method of simultaneous displacements is structurally well suited to parallel execution, its low rate of convergence makes it somewhat unattractive.

The method of successive displacements, while apparently unsuited from a structural point of view, is in fact quite attractive for parallel computation. The rate of convergence for the method of successive displacements
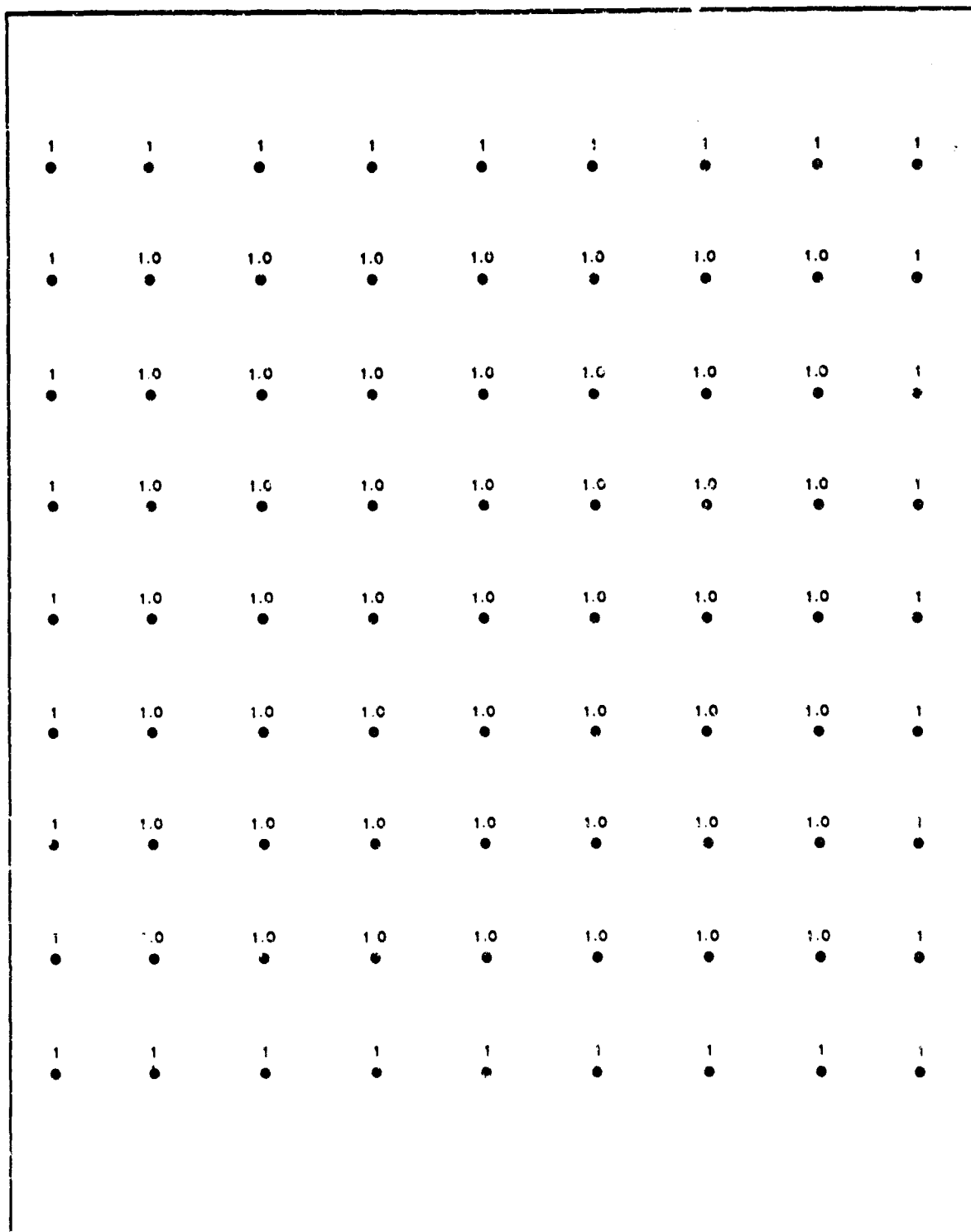
Figure VIII-11 - Parallel Fill-In, Run 1
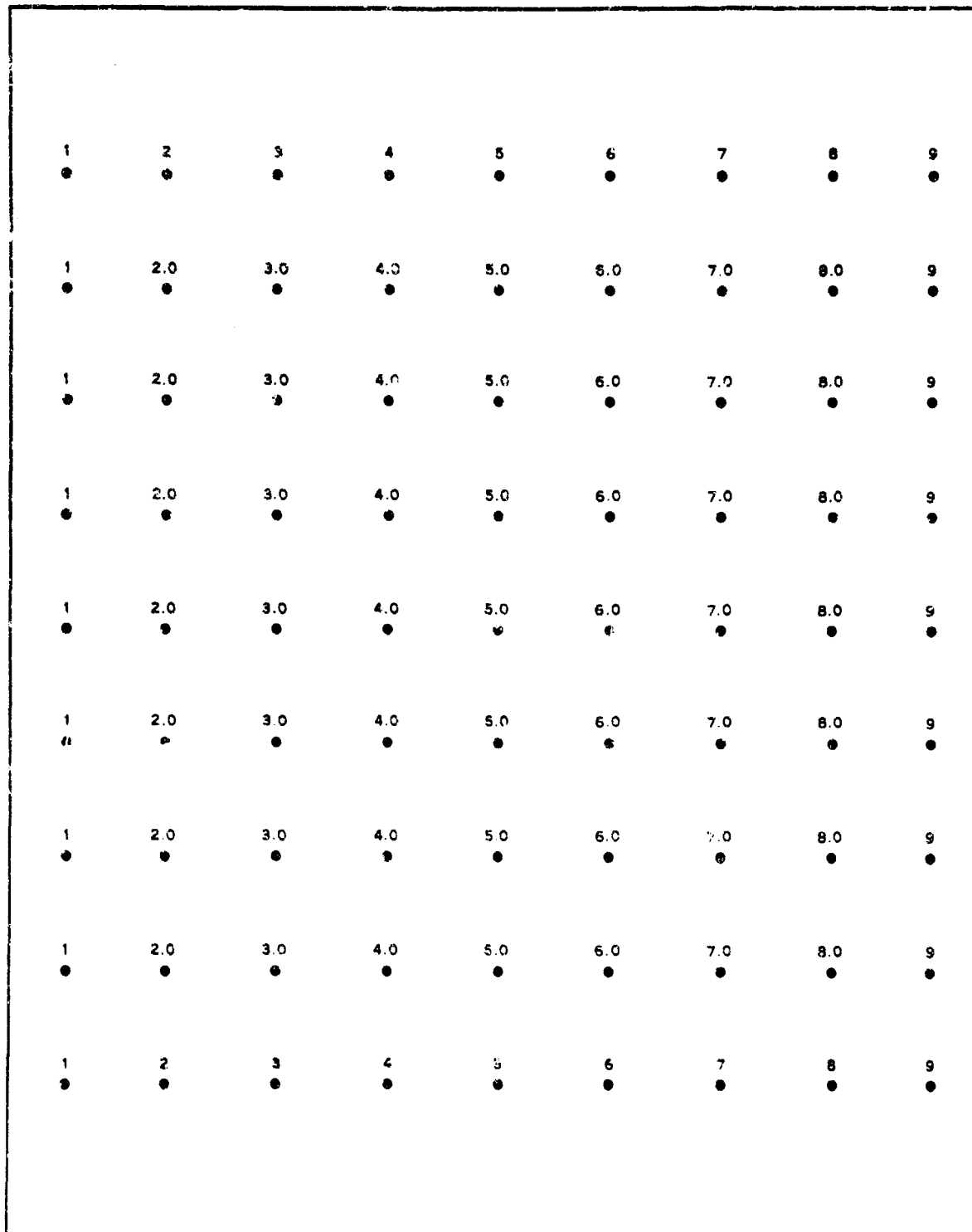
Figure VIII-12 - Parallel Fill-In, Run 2

Figure VIII-i3 - Parallel Fill-in, Run 3

| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| 8 | 7.25 | 6.50 | 5.75 | 5.00 | 4.25 | 3.50 | 2.75 | 2 |
| 7 | 6.50 | 6.00 | 5.50 | 5.00 | 4.50 | 4.00 | 3.50 | 3 |
| 6 | 5.75 | 5.50 | 5.25 | 5.00 | 4.75 | 4.50 | 4.25 | 4 |
| 5 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5 |
| 4 | 4.25 | 4.50 | 4.75 | 5.00 | 5.25 | 5.50 | 5.75 | 6 |
| 3 | 3.50 | 4.00 | 4.50 | 5.00 | 5.50 | 6.00 | 6.50 | 7 |
| 2 | 2.75 | 3.50 | 4.25 | 5.00 | 5.75 | 6.50 | 7.25 | 8 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Figure VIII-14 - Parallel Fill-In, Run 4

| 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| ● | ● | ● | ● | ● | ● | ● | ● | ● |

| 1 | 1.88 | 2.70 | 3.44 | 3.78 | 3.44 | 2.70 | 1.88 | 1 |
|---|---|---|---|---|---|---|---|---|
| ● | ● | ● | ● | ● | ● | ● | ● | ● |

| 1 | 1.76 | 2.50 | 2.28 | 3.25 | 2.98 | 2.50 | 1.76 | · |
|---|---|---|---|---|---|---|---|---|
| ● | ● | ● | ● | ● | ● | ● | ● | ● |

| 1 | 1.69 | 2.30 | 2.75 | 2.94 | 2.75 | 2.30 | 1.69 | · |
|---|---|---|---|---|---|---|---|---|
| ● | ● | ● | ● | · | ● | ● | ● | · |

| 1 | 1.66 | 2.25 | 2.69 | 3.00 | 2.69 | 2.25 | 1.66 | 1 |
|---|---|---|---|---|---|---|---|---|
| ● | ● | ● | ● | o | ● | ● | ● | · |

| 1 | 1.69 | 2.30 | 2.75 | 2.94 | 2.75 | 2.30 | 1.69 | 1 |
|---|---|---|---|---|---|---|---|---|
| ● | ● | ● | ● | ● | ● | ● | ● | ● |

| 1 | 1.76 | 2.50 | 2.98 | 3.25 | 2.98 | 2.50 | 1.76 | 1 |
|---|---|---|---|---|---|---|---|---|
| ● | ● | ● | ● | ● | ● | ● | ● | ● |

| 1 | 1.88 | 2.70 | 3.44 | 3.78 | 3.44 | 3.70 | 1.88 | 1 |
|---|---|---|---|---|---|---|---|---|
| ● | ● | ● | ● | ● | ● | ● | ● | ● |

| 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| ● | ● | ● | ● | ● | ● | ● | ● | ● |

Figure VIII-15 · Parallel Fill-In. Run 5

| 1 | $\sqrt{2}$ | $\sqrt{3}$ | $\sqrt{4}$ | $\sqrt{5}$ | $\sqrt{6}$ | $\sqrt{7}$ | $\sqrt{8}$ | 3 |
|---|---|---|---|---|---|---|---|---|
| • | • | • | • | • | • | • | • | • |
| 1 | 1.33 | 1.64 | 1.92 | 2.17 | 2.40 | 2.61 | 2.81 | 3 |
| • | • | • | • | • | • | • | • | • |
| 1 | 1.30 | 1.59 | 1.87 | 2.13 | 2.37 | 2.59 | 2.8C | 3 |
| • | • | • | • | • | • | • | • | • |
| 1 | 1.29 | 1.58 | 1.85 | 2.11 | 2.35 | 2.58 | 2.79 | 3 |
| • | • | • | • | • | • | • | • | • |
| 1 | 1.29 | 1.57 | 1.85 | 2.12 | 2.35 | 2.57 | 2.79 | 3 |
| • | • | • | • | • | • | • | • | • |
| 1 | 1.29 | 1.58 | 1.85 | 2.11 | 2.35 | 2.58 | 2.79 | 3 |
| • | • | • | • | • | • | • | • | • |
| 1 | 1.30 | 1.59 | 1.87 | 2.13 | 2.37 | 2.59 | 2.80 | 3 |
| • | • | • | • | • | • | • | • | • |
| 1 | 1.33 | 1.54 | 1.92 | 2.17 | 2.40 | 2.61 | 2.81 | 3 |
| • | • | • | • | • | • | • | • | • |
| 1 | $\sqrt{2}$ | $\sqrt{3}$ | $\sqrt{4}$ | $\sqrt{5}$ | $\sqrt{6}$ | $\sqrt{7}$ | $\sqrt{8}$ | 3 |
| • | • | • | • | • | • | • | • | • |

Figure VIII-16 - Parallel Fill-In, Run 6

-214-

| $1$ | $\sqrt{2}$ | $\sqrt{3}$ | $\sqrt{4}$ | $\sqrt{5}$ | $\sqrt{4}$ | $\sqrt{3}$ | $\sqrt{2}$ | $1$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1.30 | 1.57 | 1.78 | 1.87 | 1.78 | 1.57 | 1.30 | 1 |
| 1 | 1.24 | 1.46 | 1.62 | 1.70 | 1.62 | 1.46 | 1.24 | 1 |
| 1 | 1.21 | 1.40 | 1.54 | 1.60 | 1.54 | 1.40 | 1.21 | 1 |
| 1 | 1.20 | 1.39 | 1.52 | 1.62 | 1.52 | 1.39 | 1.20 | 1 |
| 1 | 1.21 | 1.40 | 1.54 | 1.60 | 1.54 | 1.40 | 1.21 | 1 |
| 1 | 1.24 | 1.46 | 1.62 | 1.70 | 1.62 | 1.46 | 1.24 | 1 |
| 1 | 1.30 | 1.57 | 1.78 | 1.87 | 1.78 | 1.57 | 1.30 | 1 |
| $1$ | $\sqrt{2}$ | $\sqrt{3}$ | $\sqrt{4}$ | $\sqrt{5}$ | $\sqrt{4}$ | $\sqrt{3}$ | $\sqrt{2}$ | $1$ |

Figure VIII-17 - Parallel Fill-In, Run 7

Figure VIII-18 - Parallel Fill-In, Run 8

| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| -1 | 0.00 | 0.39 | 0.56 | 0.59 | 0.56 | 0.39 | 0.00 | -1 |
| -1 | -0.39 | 0.00 | 0.20 | 0.25 | 0.20 | 0.00 | -0.39 | -1 |
| -1 | -0.56 | -0.20 | 0.00 | 0.06 | 0.00 | -0.20 | -0.59 | -1 |
| -1 | -0.59 | -0.25 | -0.06 | 0.00 | -0.06 | -0.25 | -0.59 | -1 |
| -1 | -0.56 | -0.20 | 0.00 | 0.06 | 0.00 | -0.20 | -0.56 | -1 |
| -1 | -0.39 | 0.00 | 0.20 | 0.25 | 0.20 | 0.00 | -0.39 | -1 |
| -1 | 0.00 | 0.39 | 0.56 | 0.59 | 0.56 | 0.39 | 0.00 | -1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
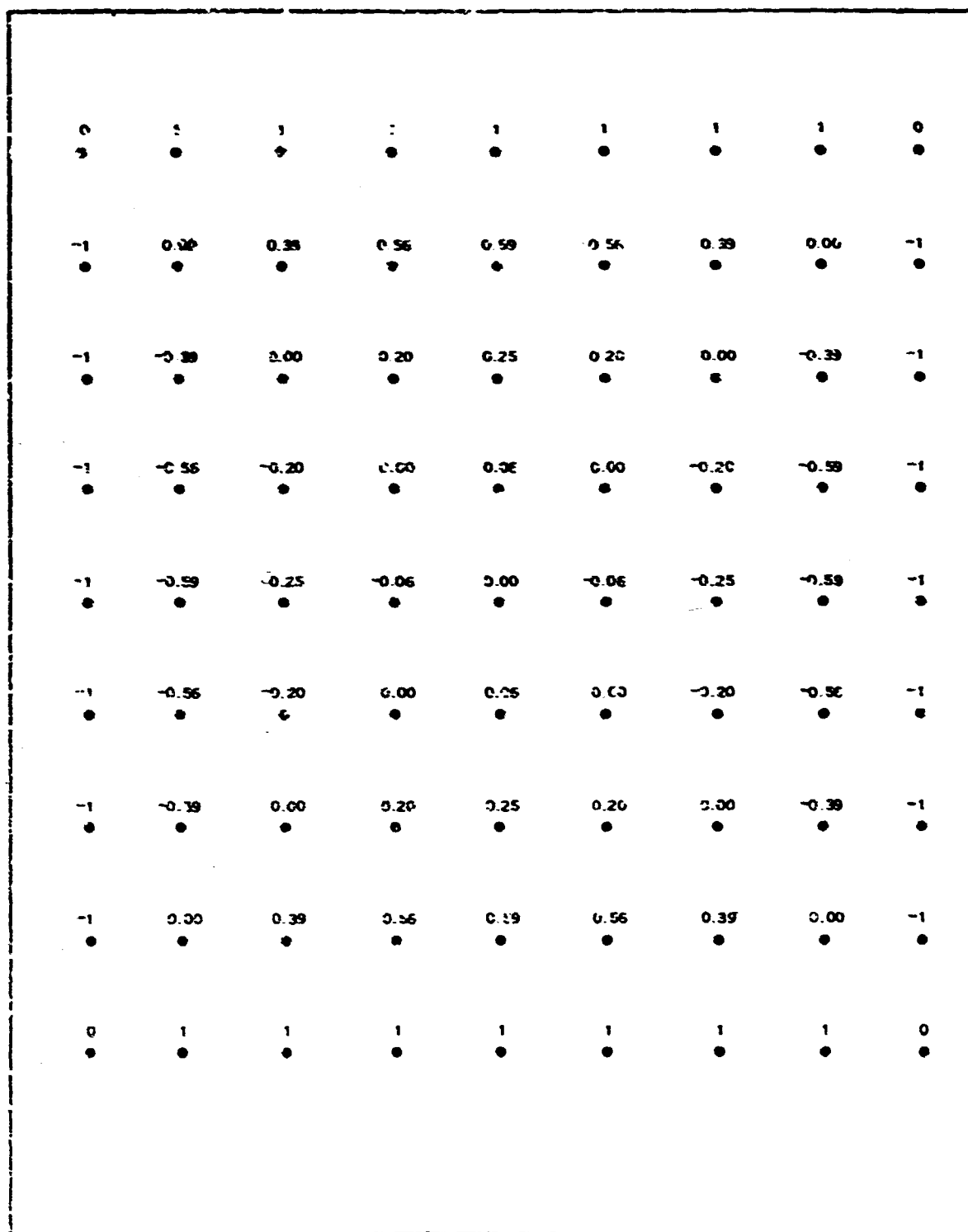
Figure VIII-19 - Parallel Fill-In, Run 9

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1.5 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2.5 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 1 | 1.53 | 1.76 | 1.91 | 2.00 | 2.09 | 2.23 | 2.47 | 3 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 1 | 1.38 | 1.62 | 1.83 | 2.00 | 2.17 | 2.38 | 2.64 | 3 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 1 | 1.30 | 1.57 | 1.80 | 2.00 | 2.20 | 2.43 | 2.70 | 3 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 1 | 1.29 | 1.56 | 1.79 | 2.00 | 2.20 | 2.44 | 2.71 | 3 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 1 | 1.30 | 1.57 | 1.80 | 2.00 | 2.20 | 2.43 | 2.70 | 3 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 1 | 1.36 | 1.62 | 1.83 | 2.00 | 2.17 | 2.38 | 2.64 | 3 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 1 | 1.53 | 1.76 | 1.91 | 2.00 | 2.09 | 2.23 | 2.47 | 3 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 1.5 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2.5 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |

Figure VIII-20 - Parallel Fill-In, Run 10

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0.5 | √2 | √3 | √4 | √5 | √4 | √3 | √2 | 1 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 0 | 0.76 | 1.20 | 1.50 | 1.63 | 1.59 | 1.44 | 1.23 | 1 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 0 | 0.47 | 0.82 | 1.06 | 1.20 | 1.23 | 1.20 | 1.10 | 1 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 0 | 0.28 | 0.53 | 0.72 | 0.85 | 0.92 | 0.96 | 0.96 | 1 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 0 | 0.14 | 0.29 | 0.42 | 0.55 | 0.63 | 0.72 | 0.86 | 5 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 0 | 0.01 | 0.04 | 0.12 | 0.22 | 0.32 | 0.48 | 0.72 | 1 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 0 | −0.16 | −0.24 | −0.21 | −0.13 | −0.04 | 0.14 | 0.47 | 1 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| 0 | −0.43 | −0.56 | −0.59 | −0.56 | −0.50 | −0.33 | 0.03 | 1 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
| −0.5 | −1 | −1 | −1 | −1 | −1 | −1 | −1 | 1 |
| ● | ● | ● | ● | ● | ● | ● | ● | ● |
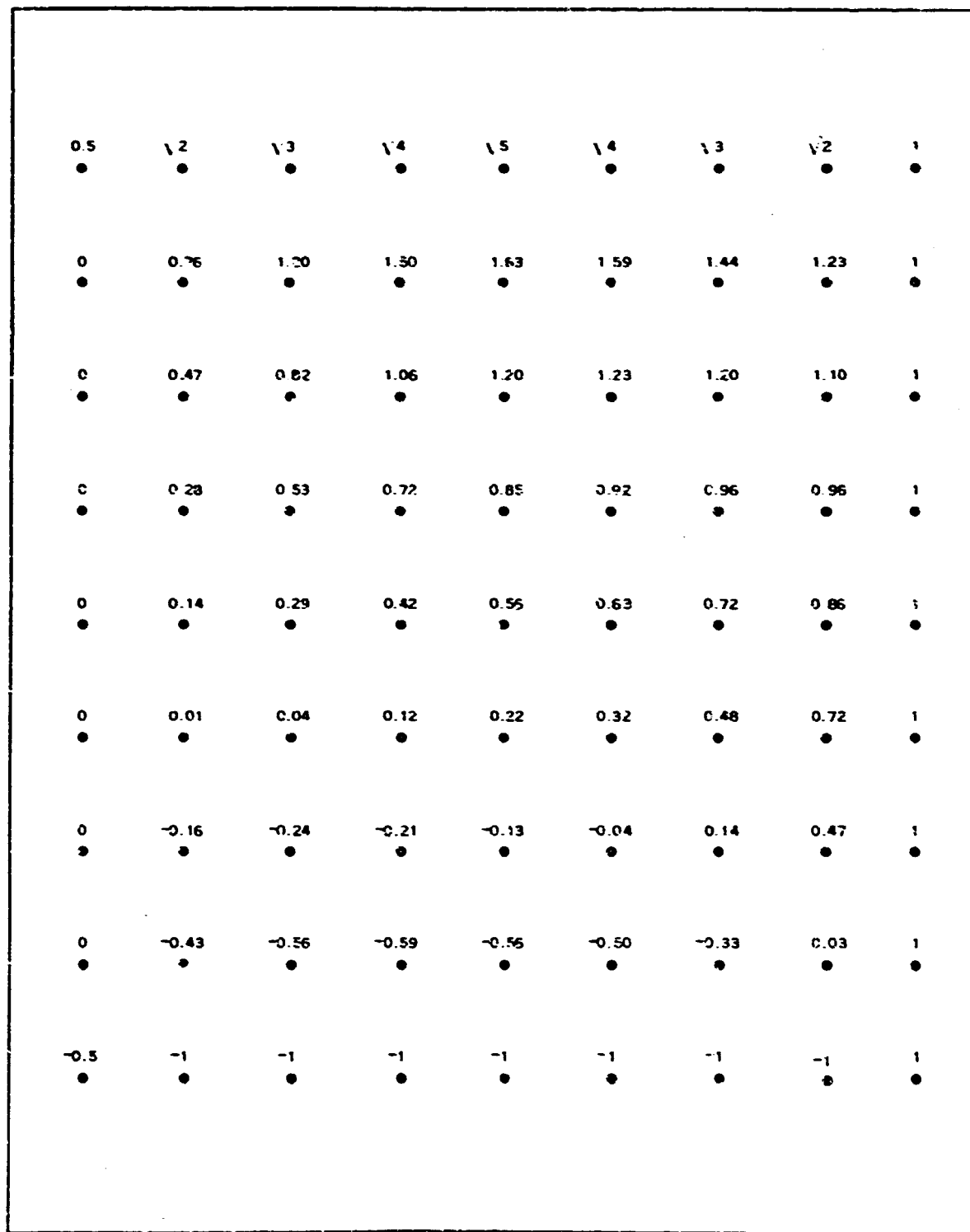
Figure VIII-21 - Parallel Fill-In, Run 11

-219-

can be improved by the use of acceleration parameters. Further, since in practice the number of internal mesh points involved in the solution of Laplace's equation will greatly exceed the number of processing units available on a parallel processor, a "parallel by block, sequential by point within a block" type iteration must be used and such an iteration is well suited to the method of successive displacements.

The PFI method for obtaining initial approximations to Laplace's equation over the interior points of a mesh is ideally suited to parallel execution.

A test for convergence based on maximum pointwise change in approximation values between successive iterations could be accomplished readily on Machines I or II due to the rapid sort capability.

## 5. CONCLUSIONS

This appendix has reviewed several mathematical techniques and analyzed their suitability for parallel execution. These techniques are Jacobi's method for the determination of eigenvalues of real symmetric matrices, the relaxation solution to a system of linear algebraic equations, numerical solution to Laplace's equation, and mesh fill in. Each technique was seen to be amenable to parallel execution. It was further seen that each technique involved searching a set for the element of maximum magnitude, a process well suited to a machine having sorting capability.

The inherent parallelism resident in each of the techniques provides a suitable basis for a study to determine optimal methods of parallel execution.

## 6. REFERENCES

1. Faddeev, D. K., and Faddeeva, V. N.: Computational Methods of Linear Algebra. San Francisco, Calif., W. H. Freeman and Co., 1963.

2. Beckenbaugh, E. F.: Modern Mathematics for the Engineer. New York, N. Y., McGraw-Hill, 1965.

3.  Forsythe, G. E., and Wasow, W. R.: Finite Difference Methods for Partial Differential Equations. New York, N.Y., John Wiley and Sons, 1959.

4.  Varga, R. S.: Matrix Iterative Analysis. Englewood Cliffs, N.J., Prentice-Hall, 1962.

5.  Macon, N.: Numerical Analysis. New York, N.Y., John Wiley and Sons, 1963.

6.  Householder, A. S.: The Theory of Matrices in Numerical Analysis. New York, N.Y., Blaisdell Publishing Co., 1965.

## 1. INTRODUCTION

Concurrent with efforts directed toward the design and efficient utilization of parallel processors has been the realization that computational capabilities resident in parallel processors give rise to new ways of thinking about problems, their fundamental structure, and appropriate solution models. It is therefore desirable that macro machine instructions, in fact a programming language, be developed that allow and indeed promote ease of conceiving and expressing the structure of parallel solution models. This appendix presents a brief list of instructions capable of compactly representing operations within a parallel solution model. Computational examples are given along with a suggestion for expanding and generalizing the instructions into a programming language.

## 2. DEFINITIONS

Let the Greek letters $\alpha$, $\beta$, $\gamma$, . . . . denote vectors of the form

$$\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \tag{1}$$

where $\alpha_i$ (i = 1, 2, . . . , n) is a real number unless otherwise specified. In expressions such as

$$\alpha = (\alpha_1, \alpha_2, \ldots)_n, \tag{2}$$

the subscript n means that $\alpha$ is to be considered an n vector.

## 3. INSTRUCTIONS

### a. General

In the following instructions, $\alpha$, $\beta$, $\gamma$ are as defined in (1) and f denotes a

real number. The elements $\alpha_i$, $i = 1, 2, \ldots, n$ of a vector $\alpha = (\alpha_1, \alpha_2, \ldots)_n$ correspond to real numbers stored in a parallel processor. In general, an instruction will specify the execution by the parallel processor of some rule of assignment, $\Gamma$, that associates with the vector(s) $\alpha$ a vector $\gamma$. For example:

1. Suppose there is a vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ and the vector $\gamma = 2\alpha = (2\alpha_1, 2\alpha_2, \ldots, 2\alpha_n) = (\gamma_1, \gamma_2, \ldots, \gamma_2)$ is desired. Then an instruction is specified directing the parallel processor to effect the following rule of assignment:

$$\Gamma: \alpha \longrightarrow 2\alpha = \gamma .$$

2. Suppose there are vectors $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ and $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$ and the vector $\gamma = \alpha + \beta = (\alpha_1 + \beta_1, \alpha_2 + \beta_2, \ldots, \alpha_n + \beta_n) = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ is desired. Then an instruction is specified directing the parallel processor to effect the following rule of assignment:

$$\Gamma: (\alpha, \beta) \longrightarrow \alpha + \beta = \gamma .$$

b. List of Instructions

A list of instructions follows. They are designed primarily to specify the parallel execution of common arithmetic operations frequently encountered in computational procedures.

1. Shift right/left: $(\longrightarrow, t)(\alpha)/(\longleftarrow, t)(\alpha)$. This instruction operates on a single vector, $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, to produce a vector $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ under the rule of assignment:

$$\gamma_i = \begin{cases} 0 \text{ for } 1 \leq i \leq t \\ \alpha_{i-t} \text{ for } t < i \geq n \end{cases} \text{ for } (\longrightarrow, t)(\alpha) ,$$

and

-224-

$$\gamma_i = \begin{cases} 0 \text{ for } (n-t) < i \leq n \\ \alpha_{i+t} \text{ for } 1 \leq i \leq (n-t) \end{cases} \quad \text{for } (\leftarrow, t) (\alpha) .$$

Note: $\alpha$ is unchanged;[a] any overflow is lost.

Example: Let $\alpha = (1, 3, 5, 7)$. Then

$$\gamma = (\rightarrow, 2) (\alpha) = (0, 0, 1, 3) ,$$

$$\gamma = (\leftarrow, 1) (\alpha) = (3, 5, 7, 0) .$$

2. Shift right/left one: $(1\rightarrow, t) (\alpha)/(\leftarrow 1, t) (\alpha)$. This instruction operates on a single vector, $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, to produce a vector $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ under the rule of assignment:

$$\gamma_i = \begin{cases} 1 \text{ for } 1 \leq i \leq t \\ \alpha_{i-t} \text{ for } t < i \leq n \end{cases} \quad \text{for } (1\rightarrow, t) (\alpha) ,$$

and

$$\gamma_i = \begin{cases} 1 \text{ for } (n-t) < i \leq n \\ \alpha_{i+t} \text{ for } 1 < i \leq (n-t) \end{cases} \quad \text{for } (\leftarrow 1, t) (\alpha) .$$

Note: $\alpha$ is unchanged; any overflow is lost.

Example: Let $\alpha = (7, 9, 1, 8, 5)$. Then

$$\gamma = (1\rightarrow, 3) (\alpha) = (1, 1, 1, 7, 9) .$$

$$\gamma = (\leftarrow 1, 4) (\alpha) = (5, 1, 1, 1, 1) .$$

3. Spread right/left: $\langle j\rightarrow, t \rangle (\alpha)/\langle \leftarrow j, t \rangle (\alpha)$. This instruction operates on a single vector, $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, to produce a vector $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ under the following rule of assignment:

---

[a] In this and the following instructions where "$\alpha$ is unchanged," if $\alpha = \gamma$ [for example, $\alpha = (\rightarrow, t) (\alpha)$], the positions of $\alpha$ are assigned new values under the rule of assignment for the instruction.

$$\gamma_i = \begin{cases} \alpha_i \text{ for } 1 \le i < j \\ \alpha_j \text{ for } j \le i \le (j+t) \\ \alpha_i \text{ for}(j+t) < i \le n \end{cases} \text{ for } \langle j{\to}, \ t \rangle (\alpha) \ ,$$

and

$$\gamma_i = \begin{cases} i \text{ for } 1 \le i < (j-t) \\ j \text{ for}(j-t) \le i \le j \\ i \text{ for } j < i \le n \end{cases} \text{ for } \langle {\leftarrow}j, \ t \rangle (\alpha) \ .$$

Note: $\alpha$ is unchanged; any overflow is lost.

Example: Let $\alpha = (7, 9, 1, 8, 5)$. Then

$$\gamma = \langle 2{\to}, \ 2 \rangle (\alpha) = (7, 9, 9, 9, 5) \ ,$$

$$\gamma = \langle {\leftarrow}3, \ 4 \rangle (\alpha) = (1, 1, 1, 8, 5) \ .$$

Note: overflow occurs in the above example.

4.  Rotate right/left: $(RR, \ t) (\alpha)/(RL, \ t) (\alpha)$. This instruction operates on a single vector, $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n)$, altering it as follows. The elements of $\alpha$ are shifted right/left $t$ positions. Overflow out the right/left is added in on the left/right.

    Example: Let $\alpha = (7, 9, 1, 8, 5)$. Then

    $$(RR, \ 2) (\alpha) = (8, 5, 7, 9, 1) {\to} \alpha \ ,$$

    $$(RL, \ 3) (\alpha) = (8, 5, 7, 9, 1) {\to} \alpha \ .$$

5.  Set sign plus/minus: $\boxed{SSP} (\alpha)/\boxed{SSM} (\alpha)$. This instruction operates on a single vector, $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n)$, altering it as follows. Each element $\alpha_i$ of $\alpha$ is set to $|\alpha_i|/ - |\alpha_i|$. Example: Let $\alpha = (-1, 0, 7, -4, 12)$. Then

$$\boxed{\text{SSP}} \ (\alpha) \ = \ (1, \ 0, \ 7, \ 4, \ 12) \rightarrow \alpha \ ,$$

$$\boxed{\text{SSM}} \ (\alpha) \ = \ (-1, \ 0, \ -7, \ -4, \ -12) \rightarrow \alpha \ .$$

6. Scalar add/subtract/multiply/divide: $\boxed{+, \ f} / \boxed{-, \ f} /$
$\boxed{x, \ f} / \boxed{\div, \ f} \ (\alpha)$. This instruction operates on a
single vector, $\alpha = (\alpha_1, \ \alpha_2, \ \ldots, \ \alpha_n)$, to produce
a vector $\gamma = (\gamma_1, \ \gamma_2, \ \ldots, \ \gamma_n)$ under the rule of
assignment:

$$\gamma_i \ = \ (\alpha_i + f)/(\alpha_i - f)/(\alpha_i \times f)/(\alpha_i/f) \ .$$

Note: $\alpha$ is unchanged.

Example: Let $\alpha = (7, \ 9, \ 1, \ 8, \ 5)$, $f = 3$. Then

$$\gamma \ = \ \boxed{+, \ f} \ (\alpha) \ = \ (10, \ 12, \ 4, \ 11, \ 8) \ ,$$

$$\gamma \ = \ \boxed{-, \ f} \ (\alpha) \ = \ (4, \ 6, \ -2, \ 5, \ 2) \ ,$$

$$\gamma \ = \ \boxed{x, \ f} \ (\alpha) \ = \ (21, \ 27, \ 3, \ 24, \ 5) \ ,$$

$$\gamma \ = \ \boxed{\div, \ f} \ (\alpha) \ = \ (7/3, \ 3, \ 1/3, \ 8/3, \ 5/3) \ .$$

7. Vector add/subtract/multiply/divide: $\oplus/\ominus/\otimes/\oplus$
$(\alpha, \ \beta)$. This instruction operates on an ordered
pair of vectors $(\alpha, \ \beta)$:

$$\alpha \ = \ (\alpha_1, \ \alpha_2, \ \ldots, \ \alpha_n)$$

$$\beta \ = \ (\beta_1, \ \beta_2, \ \ldots, \ \beta_n) \ ,$$

to produce a vector

$$\gamma \ = \ (\gamma_1, \ \gamma_2, \ \ldots, \ \gamma_n)$$

under the following rule of assignment:

$$\gamma_i \ = \ (\alpha_i + \beta_i)/(\alpha_i - \beta_i)/(\alpha_i\beta_i)/(\alpha_i/\beta_i)$$

Note: $\alpha$ and $\beta$ are unchanged.

Example: Let $\alpha$ = (7, 9, 1, 8, 5), $\beta$ = (2, -3, 1/3, -16, 1). Then

$$\oplus (\alpha, \beta) = (9, 6, 4/3, -8, 6) ,$$

$$\ominus (\alpha, \beta) = (5, 12, 2/3, 24, 4) ,$$

$$\otimes (\alpha, \beta) = (14, -27, 1/3, -128, 5) ,$$

$$\oplus (\alpha, \beta) = (7/2, -3, 3, -1/2, 5) ,$$

8. Sum: $\Sigma(\alpha)$. This instruction effectively is a subroutine. It operates on a single vector, $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, to produce a 1-vector $\gamma = (\gamma_1)$ under the following rule of assignment:

$$\gamma_1 = \sum_{i = 1}^{n} \alpha_i .$$

Note: $\alpha$ is unchanged.

Example: Let $\alpha$ = (7, 9, 1, 8, 5). Then

$$\gamma = \Sigma(\alpha) = (30) .$$

9. Chain: $\pi(\alpha)$. This instruction effectively is a subroutine. It operates on a single-vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ to produce a 1-vector $\gamma = (\gamma_1)$ under the following rule of assignment:

$$\gamma_1 = \prod_{i = 1}^{n} \alpha_i .$$

Note: $\alpha$ is unchanged.

Example: Let $\alpha$ = (7, 9, 1, 8, 5). Then

$$\gamma = \pi(\alpha) = (2520) .$$

10. Create: $C(\alpha, n) (\alpha_1, \alpha_2, \ldots, \alpha_n)$. This instruction causes a vector of length n, called $\alpha$, to be stored in the parallel processor with elements $\alpha_i$ specified with the instruction.

Example:

$$C(\alpha, 7)(0, 1, 0, 0, 3\tfrac{1}{2}, 0, 0) \rightarrow \alpha.$$

11. If: IF $(\alpha, \mathcal{E})r, s, t$. This instruction specifies a transfer of program control according to the following rules:

   a. Let $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ and $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n)$ be n vectors.

   b. Let r, s, t specify locations to which program control can be transferred.

   c. Then program control will be transferred to r, s, or t according to whether $\alpha_i < \mathcal{E}_i$, $\alpha_i = \mathcal{E}_i$, $\alpha_i > \mathcal{E}_i$ for all $i = 1, 2, \ldots, n$.

4. SAMPLE PROGRAMS

a. General

Some sample programs written in terms of the instruction list are exhibited below. The existence of a "DO LOOP" type instruction is assumed.

b. Program 1

Given $X_0$, $\Delta$, n

Construct $V = (X_0, X_0 + \Delta, X_0 + 2\Delta, \ldots, X_0 + n\Delta)_{n+1}$

Define: $L = \left[\ell n_2 (n - 1)\right]$ (by $\left[X\right]$ is meant the greatest integer in X)

Procedure:

$$C(V, n + 1)(0, \Delta, 0, 0, \ldots, 0)$$

$$DO\ M\ k = 0,\ L$$

$$V^* = (\longrightarrow, 2^k)(V)$$

$$V^{**} = \left\langle 2^k + 1 \longrightarrow. 2^k \right\rangle (V)$$

M
$$V = \oplus(V^*, V^{**})$$

$$V = \left[ +, X_0 \right] (V)$$

Then

$$V = (X_0, X_0 + \Delta, X_0 + 2\Delta, \ldots, X_0 + n\Delta) .$$

Example: Let $n = 8$. Then $L = \left[ \ell n_2 (7) \right] = 2$.

The program would proceed as follows:

$$C(V, 9)(0, \Delta, 0, 0, 0, 0, 0, 0, 0)$$

Going through the DO Loop would give:

K = 0

$$V^* = (\longrightarrow, 1)(V) = (0, 0, \Delta, 0, 0, 0, 0, 0, 0)$$

$$V^{**} = \left\langle 2 \longrightarrow, 1 \right\rangle (V) = (0, \Delta, \Delta, 0, 0, 0, 0, 0, 0)$$

$$V = \oplus (V^*, V^{**}) = (0, \Delta, 2\Delta, 0, 0, 0, 0, 0, 0)$$

K = 1

$$V^* = (\longrightarrow, 2)(V) = (0, 0, 0, \Delta, 2\Delta, 0, 0, 0, 0)$$

$$V^{**} = \left\langle 3 \longrightarrow, 2 \right\rangle (V) = (0, \Delta, 2\Delta, 2\Delta, 2\Delta, 0, 0, 0, 0)$$

$$V = \oplus (V^*, V^{**}) = (0, \Delta, 2\Delta, 3\Delta, 4\Delta, 0, 0, 0, 0)$$

K = 2

$$V^* = (\longrightarrow, 4)(V) = (0, 0, 0, 0, 0, \Delta, 2\Delta, 3\Delta, 4\Delta)$$

$$V^{**} = \left\langle 5 \longrightarrow, 4 \right\rangle (V) = (0, \Delta, 2\Delta, 3\Delta, 4\Delta, 4\Delta, 4\Delta, 4\Delta, 4\Delta)$$

$$V = \oplus (V^*, V^{**}) = (0, \Delta, 2\Delta, 3\Delta, 4\Delta, 5\Delta, 6\Delta, 7\Delta, 8\Delta)$$

and finally

$$V = \left[+, \; X_0\right] (V) = (X_0, \; X_0 + \Delta, \; \ldots \; , \; X_0 + 8\Delta)$$

Program 2

Given : $\alpha = (\alpha_1, \; \alpha_2, \; \ldots \; , \; \alpha_n), \; \beta = (\beta_1, \; \beta_2, \; \ldots \; , \; \beta_n)$

Construct: $\gamma = \alpha \cdot \beta = $ the scalar product of $\alpha$ and $\beta$

Procedure:

$$V = \otimes(\alpha, \; \beta)$$

$$V = \Sigma(V)$$

Then $V = (\gamma_1)$, where

$$\gamma_1 = \sum_{i=1}^{n} \alpha_i \beta_i$$

Example:  Let $\alpha = (1, \; 3, \; 5), \; \beta = (2, \; 4, \; 6)$.

Then

$$V = \otimes(\alpha, \; \beta) = (2, \; 12, \; 30)$$

$$V = \Sigma(V) = (44)$$

Program 3

Given: $\alpha = (\alpha_1, \; \alpha_2, \; \ldots \; , \; \alpha_n), \quad \varepsilon = (\varepsilon_1, \; \varepsilon_2, \; \ldots \; , \; \varepsilon_n)$

where $\alpha_i > 0, \; \varepsilon_i > 0$ for $i = 1, \; 2, \; \ldots \; , \; n$

Construct: $\gamma = (\gamma_i, \; \gamma_2, \; \ldots \; , \; \gamma_n)$ where $\gamma_i = \sqrt{\alpha_i}$, and $\varepsilon_i$ is the convergence criterion for a Newton iteration

Comments:  A Newton iteration for finding $\sqrt{x}$ proceeds as follows:

$$g_{i+1} = \frac{1}{2}\left(\frac{x}{g_i} + g_i\right),$$

where $g_i$ denotes the $i^{th}$ approximation to $\sqrt{x}$.  The manner of determining

the initial guess, $g_0$, depends on the range of x and, in computer solutions, the manner in which a number x is stored in the machine. In the program to follow, x/2 is used as an initial guess to $\sqrt{x}$.

| Procedure | Corresponds to |
|---|---|
| $G = [x, 0.5] (\alpha)$ | $g_i = x/2$, initial guess |
| m $\quad B = \oplus(\alpha, G)$ | $x/g_i$ |
| $\theta = \oplus (B, G)$ | $x/g_i + g_i$ |
| $\gamma = [x, 0.5] (\theta)$ | $g_{i+1} = \frac{1}{2}\left(\frac{x}{g_i} + g_i\right)$ |
| $\delta = \ominus (\gamma, G)$ | $g_{i+1} - g_i$ |
| $\delta = [SSP] (\delta)$ | $\|g_{i+1} - g_i\|$ |
| IF $(\delta, \varepsilon)$ r, r, t | $\|g_{i+1} - g_i\| < \|\varepsilon$ ? |
| t $\quad G = \gamma$ | No, $(i + 1) \rightarrow i$ |
| Go to m | Iterate again |
| r $\quad$ Continue | Yes, $g_{i+1} = \sqrt{x}$ |

and then $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ where $\gamma_i = \sqrt{\alpha_i}$.

## 5. OBSERVATIONS

Some of the properties of the instructions listed above are as follows:

1.  Instructions 1 through 4 involve essentially a shifting right or left of the elements of a vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ with options of dropping overflow with corresponding fill-in by 0's, 1's, or end-around carry. The resulting vector is an n-vector $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ with elements from
$$\left| 0, 1, \{\alpha_i\} (i = 1, 2, \ldots, n) \right|$$

2.  Instructions 5 and 6 involve a specified arithmetic

operation on each of the elements of a vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$. The resulting vector is an n-vector $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ with elements specified in terms of $\{\alpha_i\}$, i = 1, 2, . . . , n and a common arithmetic operation.

3.  Instruction 7 involves an ordered pair of vectors $(\alpha, \beta)$, $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$ and a specified arithmetic operation for each of the couples $(\alpha_i, \beta_i)$, i = 1, 2, . . . n. The resulting vector is an n-vector $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ with elements specified in terms of the couples $(\alpha_i, \beta_i)$, i = 1, 2, . . . , n and a common arithmetic operation.

4.  Instructions 8 and 9 involve a specified arithmetic operation applied to the set of elements $\{\alpha_i\}$, i = 1, 2, . . . , n of a vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$. The resulting vector $\gamma = (\gamma_1)$ is a 1-vector whose single element is specified in terms of the set $\{\alpha_i\}$, i = 1, 2, . . . , n and an arithmetic operation.

5.  Instruction 10 creates a new vector with elements specified by the programmer.

6.  Instruction 11 specifies a transfer of control, based on the results of a test.

The above observations suggest that further study of these and other instructions, yet to be defined, will produce new insights into the nature of problems, possible solution models, and notations in terms of which solution models may be written. Experience has, in fact, already shown this to be true. The instructions discussed in this report were the results of an effort to determine arithmetic operations that would be frequently encountered in machine computation, and instructions that would

compactly specify parallel execution of such operations. The list of instructions is quite short. Efforts to express parallel solution models in terms of these instructions can be expected to produce changes in instruction form and definition, suggest new instructions, and lead to the formulation of a FORTRAN type language. The execution on a parallel processor of programs written in such a language would require the construction of a compiler to translate instructions of the type listed into an efficient program of micro instructions acceptable to the processor.

6. CONCLUSIONS

Attempts to write parallel solution models and to express the operations involved in a compact notation have led to the development of a preliminary list of macro instructions for a parallel processor. Experience gained in constructing parallel solution models and writing programs for them could provide a basis for modifying presently proposed instructions and defining new ones.

The definition and modification of instructions is essentially an effort to express compactly the operations characterizing a problem and structuring possible methods of solution. Hence, it is hoped that further development of instructions will suggest new conceptual modes in which problems and possible solutions may be analyzed, and provide insights into the nature and significance of parallelism within a problem and methods for exploiting it by new computational procedures.

# APPENDIX X - PARALLEL COMPILATION

## 1. INTRODUCTION

Investigations into machine structure and parallel execution of coded routines led logically to the problem of compiling a source program in parallel. In other words, given a sequence of statements written in, say, MAD (Michigan algorithm decoder) or FORTRAN (IBM formula translating system), how can a parallel processor be used to compile the entire set of statements in parallel?

In this appendix, an algorithm for parallel compilation is developed, a method of simulation on a sequential machine is described, and the results of a simulation for a small set of replacements are presented. It is assumed that the language statements are written in MAD and that the precedence hierarchy is that of Arden, Galler, and Graham.[a,b] The MAD language was chosen because documentation on its structure is readily available and because techniques developed through MAD can be extended to other languages.

## 2. PARALLEL COMPILATION

### a. General

During the process of compilation, a sequence of statements written in a higher language such as MAD is translated into a sequence of machine language statements. The compilation process usually decomposes higher language statements into a matrix form of triples and then, from the matrix, establishes a set of machine language statements. Included in

---

[a]University of Michigan Computing Center: Michigan Algorithm Decoder. Ann Arbor, Michigan, June 1963.

[b]Arden, B.; Galler, B.; and Graham, R.: "An Algorithm for Translating Boolean Expressions," Journal of the ACM, April 1962; 9:222-239.

the compilation process is the handling of such considerations as dimension, mode, and storage allocation.

The compilation algorithm developed here deals only with the decomposition of higher language statements into triples. The statements are restricted to replacement types involving nonsubscripted variables. The previously cited precedence hierarchy is limited to the set of operators given in Table X-1.

### TABLE X-1 - PRECEDENCE HIERARCHY

| Operator | Description | Precedence |
|----------|-------------|------------|
| .ABS. | Absolute value | highest |
| .P. | Exponentiation | |
| -u | Unary minus | |
| *, / | Multiplication, division | |
| +, - | Plus, minus | |
| = | Equals (substitution) | |
| ⊢, ⊣, (, ) | Begin statement, end statement, open parenthesis, close parenthesis | Lowest |

It is further assumed that the replacement statements are stored, symbol by symbol, in an ordered list. For example, the MAD statement

$$F = A + B* . ABS. (C + D) \qquad (1)$$

is assumed to be stored in a list as:

| Index, $i$ | Item$_i$ |
|------------|----------|
| 0 | ⊢ |
| 1 | F |
| 2 | = |
| 3 | A |

(2)

| Index, i | Item$_i$ |
|:---:|:---:|
| 4 | + |
| 5 | B |
| 6 | * |
| 7 | . ABS. |
| 8 | ( |
| 9 | C |
| 10 | + |
| 11 | D |
| 12 | ) |
| 13 | + |

$$(2)$$

Later it is shown that the set of triples corresponding to (2) is just:

Triples

| | | |
|:---:|:---:|:---:|
| C | + | D |
| O | . ABS. | $R_1$ |
| B | * | $R_2$ |
| A | + | $R_3$ |
| F | = | $R_4$ |

$$(3)$$

In (3), $R_i$ denotes the resultant from the $i^{th}$ triple (row). Now (3) is read, row by row, as:

$$R_1 = C + D$$

$$R_2 = . ABS. (R_1)$$

$$R_3 = B + R_2$$

$$R_4 = A + R_3$$

$$F = R_4$$

$$= A + B* ABS. (C + D)$$

Note that the final reading is just (1).

### b. Compilation Algorithm

In parallel compilation, one tries in successive passes to examine simultaneously many statements such as (1), stored in the fashion of (2), and to form on each pass all possible triples and simplifications for the entire set of statements.

An algorithm for effecting parallel compilation is shown in Figure X-1. On each pass, the tests (operations) indicated in Figure X-1 are applied to a list such as (2). Sequences of items taken 3, 4, or 5 at a time are sought that meet certain conditions (blanks are ignored). If the indicated conditions obtain, triples are formed and/or statements are simplified as indicated.

As the structure of the flow chart in Figure X-1 indicates, the four operations may be executed concur ently; and the algorithm is capable of decomposing, in parallel, all the substitution statements of a source-language (MAD) program into a string of triples ready for final assignment (machine language). Several passes through the loop may be required, the number depending on the size and complexity of the program to be compi.

The operations indicated in Figure X-1 proceed as follows:

1. Operation 1 looks for quadruples ABCD, where A is an operator; B is either a "-u" or an ".ABS. "; C is a variable; D is an operator such that $P(D) \leqq P(B)$, where $P(X)$ denotes the precedence of X as given in Table X-1. It is assumed that B is the $\alpha^{th}$ item on the input list. Variable C is removed, and B is replaced by the variable $R_\alpha$. A triple is formed of O, B, and C, and its resultant is stored in $R_\alpha$.

2. Operation 2 looks for all quintuples ABCDE, where A, C, and E are operators; B and D are variables;
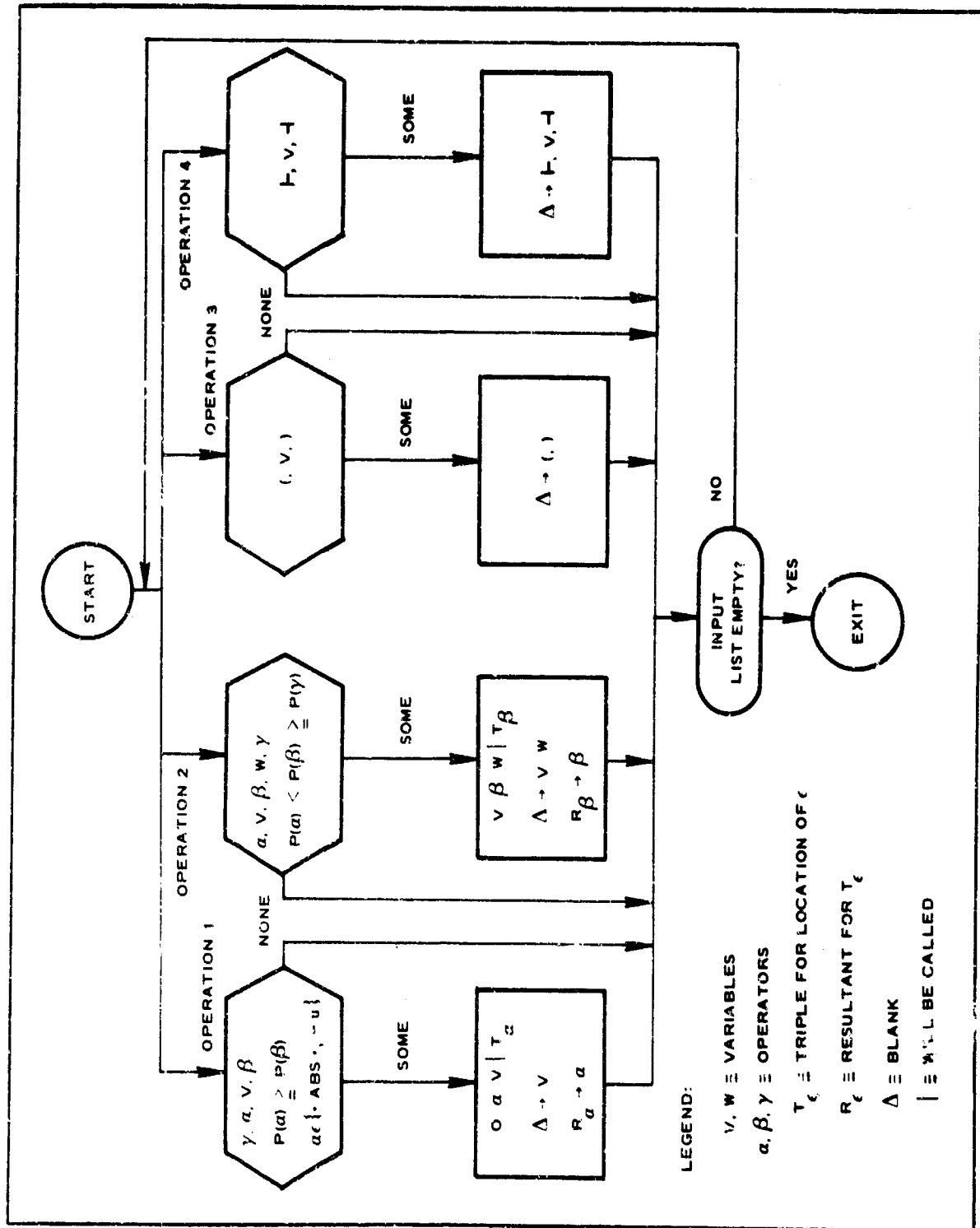
Figure X-1 - Parallel Compilation Algorithm

and $P(A) < P(C) \gtreqless P(E)$. It is assumed that C is the $\beta^{th}$ item on the input list. Variables B and D are removed, C is replaced by $R_\beta$, and a triple is formed of B, C and D, with a resultant $R_\beta$.

3. Operation 3 removes the parentheses surrounding single variables.

4. Operation 4 removes all sequences ⊢A⊣, where A is a variable.

Subsequent to the execution of these four operations, control returns to Operation 1 if the input list is not empty. Otherwise there is an exit.

In seven successive passes noted under Items $\underline{c}$ through $\underline{i}$, below, the compilation algorithm is applied to the statement (1), as stored in the list (2), and the set of triples is developed. In each of these passes, the procedure is to work through the scheme detailed in Figure X-1.

$\underline{c}$. **Pass 1**

For Operation 1, the quadruples γ, α, V, β do not exist, so that $P(\alpha) \gtreqless P(\beta)$, $\alpha \notin \{.\text{ABS}., -u\}$.

For Operation 2, the quintuple (C + D) fulfills the requirements of α, V, β, W, γ, where $P(\alpha) < P(\beta) \gtreqless P(\gamma)$. Hence, + is replaced by $R_{10}$, where $R_{10}$ is the triple C + D. Now C and D are removed from the list.

For Operation 3, there exists no triple (, V, ).

For Operation 4, there exists no triple ⊢, V, ⊣.

After Pass 1, the list (2) reads as follows (where Δ denotes a blank):

| Index, i | Item$_i$ |
| --- | --- |
| 0 | ⊢ |
| 1 | F |
| 2 | = |
| 3 | A |

| Index, $i$ | Item$_i$ |
|:---:|:---:|
| 4 | + |
| 5 | B |
| 6 | * |
| 7 | .ABS. |
| 8 | ( |
| 9 | $\Delta$ |
| 10 | $R_{10}$ |
| 11 | $\Delta$ |
| 12 | ) |
| 13 | ◄ |

<u>d</u>.   <u>Pass 2</u>

Only the condition specified by Operation 3, obtains, namely $(R_{10})$.   The parentheses are removed from the list.   After Pass 2, the list (2) reads:

| Index, $i$ | Item$_i$ |
|:---:|:---:|
| 0 | ⊦ |
| 1 | F |
| 2 | = |
| 3 | A |
| 4 | + |
| 5 | B |
| 6 | * |
| 7 | .ABS. |
| 8 | $\Delta$ |
| 9 | $\Delta$ |
| 10 | $R_{10}$ |
| 11 | $\Delta$ |
| 12 | $\Delta$ |
| 13 | ◄ |

**e.** **Pass 3**

Only the condition specified by Operation 1 obtains for $*$, $.ABS.$, $R_{10}$, $\dashv$. Hence, $.ABS.$ is replaced by $R_7$, where $R_7$ is the triple $O.ABS.R_{10}$. Now, $R_{10}$ is removed from the list. After Pass 3, the list (1) reads:

| Index, i | Item$_i$ |
|:--------:|:--------:|
| 0 | ⊢ |
| 1 | F |
| 2 | = |
| 3 | A |
| 4 | + |
| 5 | B |
| 6 | * |
| 7 | $R_7$ |
| 8 | Δ |
| 9 | Δ |
| 10 | Δ |
| 11 | Δ |
| 12 | Δ |
| 13 | ⊣ |

**f.** **Pass 4**

Only the condition specified by Operation 2 obtains for $+$, $B$, $*$, $R_7$, $\dashv$. Hence, $*$ is replaced by $R_6$, where $R_6$ is the triple $B*R_7$. Now $B$ and $R_7$ are removed from the list. After Pass 4, the list (1) reads:

| Index, i | Item$_i$ |
|:--------:|:--------:|
| 0 | ⊢ |
| 1 | F |
| 2 | = |
| 3 | A |

| Index, i | Item$_i$ |
|:---:|:---:|
| 4 | $+$ |
| 5 | $\triangle$ |
| 6 | $R_6$ |
| 7 | $\triangle$ |
| 8 | $\triangle$ |
| 9 | $\triangle$ |
| 10 | $\triangle$ |
| 11 | $\triangle$ |
| 12 | $\triangle$ |
| 13 | $\dashv$ |

g. **Pass 5**

Only the condition specified by Operation 2 obtains for $=$, A, $+$, $R_6$, $\dashv$.
Hence, $+$ is replaced by $R_4$, where $R_4$ is the triple $A + R_6$. Now A and
$R_6$ are removed from the list. After the Pass 5, the list (1) reads:

| Index, i | Item$_i$ |
|:---:|:---:|
| 0 | $\vdash$ |
| 1 | F |
| 2 | $=$ |
| 3 | $\triangle$ |
| 4 | $R_4$ |
| 5 | $\triangle$ |
| 6 | $\triangle$ |
| 7 | $\triangle$ |
| 8 | $\triangle$ |
| 9 | $\triangle$ |
| 10 | $\triangle$ |
| 11 | $\triangle$ |
| 12 | $\triangle$ |
| 13 | $\dashv$ |

### h.  Pass 6

Only the condition specified by Operation 2 obtains for $\vdash$, F, =, $R_4$, $\dashv$. Hence, = is replaced by $R_2$, where $R_2$ is the triple F = $R_4$.  Now F and $R_4$ are removed from the list.  After Pass 6, the list (1) reads:

| Index, i | Item$_i$ |
|:--------:|:--------:|
| 0 | $\vdash$ |
| 1 | $\Delta$ |
| 2 | $R_2$ |
| 3 | $\Delta$ |
| 4 | $\Delta$ |
| 5 | $\Delta$ |
| 6 | $\Delta$ |
| 7 | $\Delta$ |
| 8 | $\Delta$ |
| 9 | $\Delta$ |
| 10 | $\Delta$ |
| 11 | $\Delta$ |
| 12 | $\Delta$ |
| 13 | $\dashv$ |

### i.  Pass 7

Only the condition specified by Operation 4 obtains for $\vdash$, $R_2$, $\dashv$.  The list is emptied.  By now the compilation scheme has generated the following triples in the order indicated:

| Index | Triples | | |
|:-----:|:---:|:----:|:----:|
| $R_{10}$ | C | + | D |
| $R_7$ | O | .ABS. | $R_{10}$ |
| $R_6$ | B | * | $R_7$ |
| $R_4$ | A | $\div$ | $k_6$ |
| $R_2$ | F | = | $R_4$ |

This is effectively (3) and is read as:

$$F = R_4$$
$$= A + R_6$$
$$= A + B*R_7$$
$$= A + B*. ABS. R_{10}$$
$$= A + B*. ABS. (C + D)$$

## i. Conclusion

In this example, only one replacement statement was compiled. The compilation scheme is intended to compile many replacement statements simultaneously, and the relative speed advantage of parallel over sequential compilation increases, within machine capacity, with the number (and complexity) of statements to be compiled.

## 3. SIMULATION MODEL AND RESULTS

The compiler algorithm developed under Item 2 is designed to be implemented on a parallel processor. Since no such machine is available for checking out the algorithm on a sample problem, simulation of parallel compilation must be effected on a sequential machine. To effect the simulation, code PARCOM (parallel compilation) was written in FORTRAN IV for the IBM 1410. Code PARCOM executes the compilation algorithm in an effectively parallel fashion on a given set of replacement statements.

Code PARCOM operates as follows:

1. A sequence of replacement statements, such as
   (1), is read into the machine.

2. The symbols comprising the statements are examined and classified as variables, operators, or blanks; and precedences are assigned to the operators.

  3.   Tests specified by operations 1, 2, 3, and 4 in Figure X-1 are applied, in an effectively parallel fashion, to each of the replacement statements.

  4.   Based on the results of the tests, triples are formed and/or statement simplifications are made.

  5.   Steps 3 and 4 are repeated as necessary.

After each pass through the set of statements, code PARCOM prints out the triples formed during the pass and the resultant set of statements.

To check the algorithm, the following set of seven replacement statements was selected:

| Number | Statement |
|--------|-----------|
| 1 | $F = D + X*\left[C + X*(B + A*X)\right] + R. P. . ABS. - S$ |
| 2 | $G = -B + (B*B - 4*A*C). P. 2 /(2*A)$ |
| 3 | $H = (A + B). P. \left[C*(D + E*P/. ABS. X)\right]$ |
| 4 | $I = A + B + C - D*E$ |
| 5 | $U = F + G$ |
| 6 | $V = H*I$ |
| 7 | $W = -. ABS. (U. P. V)$ |

(4)

The results of applying the compilation algorithm to the set (4) are presented in Tables X-2 through X-13. These tables show (4) initially in the form of the list (2) and the results of successive passes. Table X-14 shows the entire list of triples generated.

As an example of how the triples represent the replacement statements, selected from Table X-14 are those triples into which Statement 1 was decomposed, namely:

| Index | Triple | | |
|-------|--------|----|----------|
| 1  | A        | *      | X         |
| 2  | O        | -      | S         |
| 15 | B        | +      | $R_1$     |
| 16 | O        | . ABS. | $R_2$     |
| 22 | R        | . P.   | $R_{16}$  |
| 27 | X        | *      | $R_{15}$  |
| 30 | C        | +      | $R_{27}$  |
| 35 | X        | *      | $R_{30}$  |
| 37 | D        | +      | $R_{35}$  |
| 40 | $R_{37}$ | +      | $R_{22}$  |
| 42 | F        | =      | $R_{40}$  |

These combine as

$$F = R_{40}$$
$$= R_{37} + R_{22}$$
$$= D + R_{35} + R. P. R_{16}$$
$$= D + X*R_{30} + R. P. . ABS. R_2$$
$$= D + X*(C + R_{27}) + R. P. . ABS. - S$$
$$= D + X*(C + X*R_{15}) + R. P. . ABS. - S$$
$$= D + X*\left[C + X*(B + R_1)\right] + R. P. . ABS. - S$$
$$= D + X*\left[C + X*(B + A*X)\right] + R. P. . ABS. - S$$

The last statement is Statement 1 from the set (4).

The triples generated on each pass correspond to basic arithmetic oper-
ations that can be performed at the time of the pass. Hence, the compila-
tion algorithm generates triples suitable for parallel execution and pro-
vides a first approach to the recognition of low level parallelism within a
source program.

## TABLE X-2 - REPLACEMENT STATEMENT SET

| | Resulting statement | | | | | | |
|---|---|---|---|---|---|---|---|
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ⊢ | ⊢ | ⊢ | ⊢ | ⊢ | ⊢ | ⊢ |
| 2 | F | G | H | I | U | V | W |
| 3 | = | = | = | = | = | = | = |
| 4 | D | ( | ( | A | F | H | - |
| 5 | + | .. | A | + | + | * | . ABS. |
| 6 | X | B | + | B | G | I | ( |
| 7 | * | + | B | + | ◄ | ┤ | U |
| 8 | ( | ( | ) | C | | | . P. |
| 9 | C | B | . P. | - | | | V |
| 10 | + | * | ( | D | | | ) |
| 11 | X | B | C | * | | | ◄ |
| 12 | * | - | * | E | | | |
| 13 | ( | 4 | ( | ◄ | | | |
| 14 | B | * | D | | | | |
| 15 | + | A | + | | | | |
| 16 | A | * | E | | | | |
| 17 | * | C | * | | | | |
| 18 | X | ) | P | | | | |
| 19 | ) | . P. | / | | | | |
| 20 | ) | 2 | . ABS. | | | | |
| 21 | + | ) | X | | | | |
| 22 | R | / | ) | | | | |
| 23 | . P. | ( | ) | | | | |
| 24 | . ABS. | 2 | ◄ | | | | |
| 25 | - | * | | | | | |
| 26 | S | A | | | | | |
| 27 | ◄ | ) | | | | | |
| 28 | | ◄ | | | | | |

## TABLE X-3 - RESULTS AFTER PASS 1

| Index | Triples | | | Contributing statement |
|---|---|---|---|---|
| 1 | A | • | X | 1 |
| 2 | O | . | S | 1 |
| 3 | O | . | B | 2 |
| 4 | B | • | B | 2 |
| 5 | 4 | • | A | 2 |
| 6 | 2 | • | A | 2 |
| 7 | A | + | B | 3 |
| 8 | E | • | P | 3 |
| 9 | O | .ABS. | S | 3 |
| 10 | A | + | B | 4 |
| 11 | D | • | E | 4 |
| 12 | F | + | G | 5 |
| 13 | H | • | I | 6 |
| 14 | U | .P. | V | 7 |

| Index | Resulting statements | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ► | ► | ► | ► | ► | ► | ► |
| 2 | F | G | H | I | U | V | W |
| 3 | = | = | = | = | = | = | = |
| 4 | D | ( | ( | △ | △ | △ | - |
| 5 | + | $R_3$ | △ | $R_{10}$ | $R_{12}$ | $R_{13}$ | .ABS. |
| 6 | X | △ | $R_7$ | △ | △ | △ | ( |
| 7 | • | + | △ | + | ◄ | ◄ | △ |
| 8 | ( | ( | ) | C | | | $R_{14}$ |
| 9 | C | △ | .P. | - | | | △ |
| 10 | + | $R_{14}$ | ( | △ | | | ) |
| 11 | X | △ | C | $R_{11}$ | | | ◄ |
| 12 | • | . | • | △ | | | |
| 13 | ( | △ | ( | ◄ | | | |
| 14 | B | $R_3$ | D | | | | |
| 15 | + | △ | + | | | | |
| 16 | △ | • | △ | | | | |
| 17 | $R_1$ | C | $R_8$ | | | | |
| 18 | △ | ) | △ | | | | |
| 19 | ) | .P. | / | | | | |
| 20 | ) | 2 | $R_9$ | | | | |
| 21 | + | ) | △ | | | | |
| 22 | R | / | ) | | | | |
| 23 | .P. | ( | ) | | | | |
| 24 | .ABS. | △ | ◄ | | | | |
| 25 | $R_2$ | $R_6$ | | | | | |
| 26 | △ | △ | | | | | |
| 27 | ◄ | ) | | | | | |
| 28 | | ◄ | | | | | |

NOTE: △ denotes blank.

## TABLE X-4 - RESULTS AFTER PASS 2

| Index | Triples | | | Contributing statement |
|---|---|---|---|---|
| 15 | B | + | $R_1$ | 1 |
| 16 | O | .ABS. | $R_2$ | 1 |
| 17 | $R_5$ | * | C | 2 |
| 18 | $R_8$ | / | $R_9$ | 3 |
| 19 | $R_{10}$ | + | C | 4 |
| 20 | U | * | $R_{12}$ | 5 |
| 21 | V | * | $R_{13}$ | 6 |

| Index | Resulting statements | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ► | ► | ► | ► | ► | ► | ► |
| 2 | F | G | H | I | Δ | Δ | W |
| 3 | = | = | = | = | $R_{20}$ | $R_{21}$ | = |
| 4 | D | ( | Δ | Δ | Δ | Δ | - |
| 5 | + | $R_3$ | Δ | Δ | Δ | Δ | .ABS. |
| 6 | X | Δ | $R_7$ | Δ | Δ | Δ | Δ |
| 7 | * | + | Δ | $R_{19}$ | ◄ | ◄ | Δ |
| 8 | ( | ( | Δ | Δ | | | $R_{14}$ |
| 9 | C | Δ | .P. | - | | | Δ |
| 10 | + | $R_4$ | ( | Δ | | | Δ |
| 11 | X | Δ | C | $R_{11}$ | | | ◄ |
| 12 | * | - | + | Δ | | | |
| 13 | ( | Δ | ( | ◄ | | | |
| 14 | Δ | Δ | D | | | | |
| 15 | $R_{15}$ | Δ | + | | | | |
| 16 | Δ | $R_{17}$ | Δ | | | | |
| 17 | Δ | Δ | Δ | | | | |
| 18 | Δ | ) | Δ | | | | |
| 19 | ) | .P. | $R_{18}$ | | | | |
| 20 | ) | 2 | Δ | | | | |
| 21 | + | ) | Δ | | | | |
| 22 | R | / | ) | | | | |
| 23 | .P. | Δ | ) | | | | |
| 24 | $R_{16}$ | Δ | ◄ | | | | |
| 25 | Δ | $R_6$ | | | | | |
| 26 | Δ | Δ | | | | | |
| 27 | ◄ | Δ | | | | | |
| 28 | | ◄ | | | | | |

NOTE: Δ denotes blank.

## TABLE X-5 - RESULTS AFTER PASS 3

| Index | Triples | | | Contributing statement |
|---|---|---|---|---|
| 22 | R | .P. | $R_{16}$ | 1 |
| 23 | $R_4$ | - | $R_{17}$ | 2 |
| 24 | D | + | $R_{18}$ | 3 |
| 25 | $R_{19}$ | - | $R_{11}$ | 4 |
| 26 | O | .ABS. | $R_{14}$ | 7 |

| Index | Resulting statements | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ▸ | ▸ | ▸ | ▸ | Δ | Δ | ▸ |
| 2 | F | G | H | I | Δ | Δ | W |
| 3 | = | = | = | = | Δ | Δ | = |
| 4 | D | ( | Δ | Δ | Δ | Δ | - |
| 5 | + | $R_3$ | Δ | Δ | Δ | Δ | $R_{26}$ |
| 6 | X | Δ | $R_7$ | Δ | Δ | Δ | Δ |
| 7 | * | + | Δ | Δ | Δ | Δ | Δ |
| 8 | ( | ( | Δ | Δ | | | Δ |
| 9 | C | Δ | .P. | $R_{25}$ | | | Δ |
| 10 | + | Δ | ( | Δ | | | Δ |
| 11 | X | Δ | C | Δ | | | ◂ |
| 12 | * | $R_{23}$ | * | Δ | | | |
| 13 | Δ | Δ | ( | ◂ | | | |
| 14 | Δ | Δ | Δ | | | | |
| 15 | $R_{15}$ | Δ | $R_{24}$ | | | | |
| 16 | Δ | Δ | Δ | | | | |
| 17 | Δ | Δ | Δ | | | | |
| 18 | Δ | ) | Δ | | | | |
| 19 | Δ | .P. | Δ | | | | |
| 20 | ) | : | Δ | | | | |
| 21 | + | ) | Δ | | | | |
| 22 | Δ | / | ) | | | | |
| 23 | $R_{22}$ | Δ | ) | | | | |
| 24 | Δ | Δ | ◂ | | | | |
| 25 | Δ | $R_6$ | | | | | |
| 26 | Δ | Δ | | | | | |
| 27 | ◂ | Δ | | | | | |
| 28 | | ◂ | | | | | |

NOTE: Δ denotes blank.

## TABLE X-6 - RESULTS AFTER PASS 4

| Index | Triples | | | Contributing statement |
|---|---|---|---|---|
| 27 | X | * | $R_{15}$ | 1 |
| 28 | I | = | $R_{25}$ | 4 |
| 29 | O | - | $R_{26}$ | 7 |

| Index | Resulting statements | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ▸ | ▸ | ▸ | ▸ | Δ | Δ | ▸ |
| 2 | F | G | H | Δ | Δ | Δ | W |
| 3 | = | = | = | $R_{28}$ | Δ | Δ | = |
| 4 | D | ( | Δ | Δ | Δ | Δ | $R_{29}$ |
| 5 | + | $R_3$ | Δ | Δ | Δ | Δ | Δ |
| 6 | X | Δ | $R_7$ | Δ | Δ | Δ | Δ |
| 7 | * | + | Δ | Δ | Δ | Δ | Δ |
| 8 | ( | Δ | Δ | Δ | | | Δ |
| 9 | C | Δ | .P. | Δ | | | Δ |
| 10 | + | Δ | ( | Δ | | | Δ |
| 11 | Δ | Δ | C | Δ | | | ◄ |
| 12 | $R_{27}$ | $R_{23}$ | * | Δ | | | |
| 13 | Δ | Δ | Δ | ◄ | | | |
| 14 | Δ | Δ | Δ | | | | |
| 15 | Δ | Δ | $R_{24}$ | | | | |
| 16 | Δ | Δ | Δ | | | | |
| 17 | Δ | Δ | Δ | | | | |
| 18 | Δ | Δ | Δ | | | | |
| 19 | Δ | .P. | Δ | | | | |
| 20 | ) | z | Δ | | | | |
| 21 | + | ) | Δ | | | | |
| 22 | Δ | / | Δ | | | | |
| 23 | $R_{22}$ | Δ | ) | | | | |
| 24 | Δ | Δ | ◄ | | | | |
| 25 | Δ | $R_6$ | | | | | |
| 26 | Δ | Δ | | | | | |
| 27 | ◄ | Δ | | | | | |
| 28 | | ◄ | | | | | |

NOTE: Δ denotes blank.

## TABLE X-7 - RESULTS AFTER PASS 5

| Index | Triples | | | Contributing statement |
|---|---|---|---|---|
| 30 | C | + | $R_{27}$ | 1 |
| 31 | $R_{23}$ | . P. | 2 | 2 |
| 32 | C | * | $R_{24}$ | 3 |
| 33 | W | = | $R_{29}$ | 7 |

| Index | Resulting statements | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ► | ► | ► | Δ | Δ | Δ | ► |
| 2 | F | G | H | Δ | Δ | Δ | Δ |
| 3 | = | = | = | Δ | Δ | Δ | $R_{33}$ |
| 4 | D | ( | Δ | Δ | Δ | Δ | Δ |
| 5 | + | $R_3$ | Δ | Δ | Δ | Δ | Δ |
| 6 | X | Δ | $R_7$ | Δ | Δ | Δ | Δ |
| 7 | * | + | Δ | Δ | Δ | Δ | Δ |
| 8 | ( | Δ | Δ | Δ | | | Δ |
| 9 | Δ | Δ | . P. | Δ | | | Δ |
| 10 | $R_{30}$ | Δ | ( | Δ | | • | Δ |
| 11 | Δ | Δ | Δ | Δ | | | ◄ |
| 12 | Δ | Δ | $R_{32}$ | Δ | | | |
| 13 | Δ | Δ | Δ | Δ | | | |
| 14 | Δ | Δ | Δ | | | | |
| 15 | Δ | Δ | Δ | | | | |
| 16 | Δ | Δ | Δ | | | | |
| 17 | Δ | Δ | ∴ | | | | |
| 18 | Δ | Δ | Δ | | | | |
| 19 | Δ | $R_{31}$ | Δ | | | | |
| 20 | ) | Δ | Δ | | | | |
| 21 | + | ) | Δ | | | | |
| 22 | Δ | / | Δ | | | | |
| 23 | $R_{22}$ | Δ | ) | | | | |
| 24 | Δ | Δ | ◄ | | | | |
| 25 | Δ | $R_6$ | | | | | |
| 26 | Δ | Δ | | | | | |
| 27 | ◄ | Δ | | | | | |
| 28 | | ◄ | | | | | |

NOTE: Δ denotes blank.

## TABLE X-8 - RESULTS AFTER PASS 6

| Index | Triples | | | Contributing statement | | | |
|---|---|---|---|---|---|---|---|
| 34 | $R_3$ | + | $R_{31}$ | 2 | | | |

| | Resulting statements | | | | | | |
|---|---|---|---|---|---|---|---|
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ⊢ | ⊢ | ⊢ | Δ | Δ | Δ | Δ |
| 2 | F | G | H | Δ | Δ | Δ | Δ |
| 3 | = | = | = | Δ | Δ | Δ | Δ |
| 4 | D | ( | Δ | Δ. | Δ | Δ | Δ |
| 5 | + | Δ | Δ | Δ | Δ | Δ | Δ |
| 6 | X | Δ | $R_7$ | Δ | Δ | Δ | Δ |
| 7 | * | $R_{34}$ | Δ | Δ | Δ | Δ | Δ |
| 8 | Δ | Δ | Δ | Δ | | | Δ |
| 9 | Δ | Δ | . P. | Δ | | | Δ |
| 10 | $R_{30}$ | Δ | Δ | Δ | | | Δ |
| 11 | Δ | Δ | Δ | Δ | | | Δ |
| 12 | Δ | Δ | $R_{32}$ | Δ | | | |
| 13 | Δ | Δ | Δ | Δ | | | |
| 14 | Δ | Δ | Δ | | | | |
| 15 | Δ | Δ | Δ | | | | |
| 16 | Δ | Δ | Δ | | | | |
| 17 | Δ | Δ | Δ | | | | |
| 18 | Δ | Δ | Δ | | | | |
| 19 | Δ | Δ | Δ | | | | |
| 20 | Δ | Δ | Δ | | | | |
| 21 | + | ) | Δ | | | | |
| 22 | Δ | / | Δ | | | | |
| 23 | $R_{22}$ | Δ | Δ | | | | |
| 24 | Δ | Δ | ◄ | | | | |
| 25 | Δ | $R_6$ | | | | | |
| 26 | Δ | Δ | | | | | |
| 27 | ◄ | Δ | | | | | |
| 28 | | ◄ | | | | | |

NOTE: Δ denotes blank.

## TABLE X-9 - RESULTS AFTER PASS 7

| Index | Triples | | | Contributing statement | | | |
|---|---|---|---|---|---|---|---|
| 35 | X | * | $R_{30}$ | 1 | | | |
| 36 | $R_7$ | .F. | $R_{32}$ | 3 | | | |

| Index | Resulting statements | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ► | ► | ► | Δ | Δ | Δ | Δ |
| 2 | F | G | H | Δ | Δ | Δ | Δ |
| 3 | = | = | = | Δ | Δ | Δ | Δ |
| 4 | D | Δ | Δ | Δ | Δ | Δ | Δ |
| 5 | + | Δ | Δ | Δ | Δ | Δ | Δ |
| 6 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 7 | $R_{35}$ | $R_{34}$ | Δ | Δ | Δ | Δ | Δ |
| 8 | Δ | Δ | Δ | Δ | | | Δ |
| 9 | Δ | ± | $R_{36}$ | Δ | | | Δ |
| 10 | Δ | Δ | Δ | Δ | | | Δ |
| 11 | Δ | Δ | Δ | Δ | | | Δ |
| 12 | Δ | Δ | Δ | Δ | | | |
| 13 | Δ | Δ | Δ | Δ | | | |
| 14 | Δ | Δ | Δ | | | | |
| 15 | Δ | Δ | Δ | | | | |
| 16 | Δ | Δ | Δ | | | | |
| 17 | Δ | Δ | Δ | | | | |
| 18 | Δ | Δ | Δ | | | | |
| 19 | Δ | Δ | Δ | | | | |
| 20 | Δ | Δ | Δ | | | | |
| 21 | + | Δ | Δ | | | | |
| 22 | Δ | / | Δ | | | | |
| 23 | $R_{22}$ | Δ | Δ | | | | |
| 24 | Δ | Δ | • | | | | |
| 25 | Δ | $R_6$ | | | | | |
| 26 | Δ | Δ | | | | | |
| 27 | • | Δ | | | | | |
| 28 | | • | | | | | |

NOTE: Δ denotes blank.

## TABLE X-10 - RESULTS AFTER PASS 8

| Index | Triples | | | Contributing statement | | | |
|-------|---------|---|---|------------------------|---|---|---|
| 37 | D | + | $R_{35}$ | 1 | | | |
| 38 | $R_{34}$ | / | $R_6$ | 2 | | | |
| 39 | H | $\approx$ | $R_{36}$ | 3 | | | |

| Index | Resulting statements | | | | | | |
|-------|---|---|---|---|---|---|---|
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ► | ► | ► | Δ | Δ | Δ | Δ |
| 2 | F | . G | Δ | Δ | Δ | Δ | Δ |
| 3 | $\approx$ | $\approx$ | $R_{39}$ | Δ | Δ | Δ | Δ |
| 4 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 5 | $R_{37}$ | Δ | Δ | Δ | Δ | Δ | Δ |
| 6 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 7 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 8 | Δ | Δ | Δ | Δ | | | Δ |
| 9 | Δ | Δ | Δ | Δ | | | Δ |
| 10 | Δ | Δ | Δ | Δ | | | Δ |
| 11 | Δ | Δ | Δ | Δ | | | Δ |
| 12 | Δ | Δ | Δ | Δ | | | |
| 13 | Δ | Δ | Δ | Δ | | | |
| 14 | Δ | Δ | Δ | | | | |
| 15 | Δ | Δ | Δ | | | | |
| 16 | Δ | Δ | Δ | | | | |
| 17 | Δ | Δ | Δ | | | | |
| 18 | Δ | Δ | Δ | | | | |
| 19 | Δ | Δ | Δ | | | | |
| 20 | Δ | Δ | Δ | | | | |
| 21 | + | Δ | Δ | | | | |
| 22 | Δ | $R_{38}$ | Δ | | | | |
| 23 | $R_{22}$ | Δ | Δ | | | | |
| 24 | Δ | Δ | ◄ | | | | |
| 25 | Δ | Δ | | | | | |
| 26 | Δ | Δ | | | | | |
| 27 | ◄ | Δ | | | | | |
| 28 | | ◄ | | | | | |

NOTE: Δ denotes blank.

## TABLE X-11 - RESULTS AFTER PASS 9

| Index | Triples | | | Contributing statement |
|---|---|---|---|---|
| 40 | $R_{37}$ | + | $R_{22}$ | 1 |
| 41 | G | = | $R_{38}$ | 2 |

| Index | Resulting statements | | | | | | |
|---|---|---|---|---|---|---|---|
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ► | ► | Δ | Δ | Δ | Δ | Δ |
| 2 | F | Δ | Δ | Δ | Δ | Δ | ∠ |
| 3 | = | $R_{41}$ | Δ | Δ | Δ | Δ | Δ |
| 4 | Δ | Δ | Δ | Δ | ∧ | Δ | Δ |
| 5 | Δ | Δ | Δ | Δ | ∠ | Δ | Δ |
| 6 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 7 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 8 | Δ | Δ | Δ | Δ |  |  | Δ |
| 9 | Δ | Δ | Δ | Δ |  |  | Δ |
| 10 | Δ | Δ | Δ | Δ |  |  | Δ |
| 11 | Δ | Δ | Δ | Δ |  |  | Δ |
| 12 | Δ | Δ | Δ | Δ |  |  |  |
| 13 | Δ | Δ | Δ | Δ |  |  |  |
| 14 | Δ | Δ | Δ |  |  |  |  |
| 15 | Δ | Δ | Δ |  |  |  |  |
| 16 | Δ | Δ | Δ |  |  |  |  |
| 17 | Δ | Δ | Δ |  |  |  |  |
| 18 | Δ | Δ | Δ |  |  |  |  |
| 19 | Δ | Δ | Δ |  |  |  |  |
| 20 | Δ | Δ | Δ |  |  |  |  |
| 21 | $R_{40}$ | Δ | Δ |  |  |  |  |
| 22 | Δ | Δ | Δ |  |  |  |  |
| 23 | Δ | Δ | Δ |  |  |  |  |
| 24 | Δ | Δ | Δ |  |  |  |  |
| 25 | Δ | Δ |  |  |  |  |  |
| 26 | Δ | Δ |  |  |  |  |  |
| 27 | ◄ | Δ |  |  |  |  |  |
| 28 |  | ◄ |  |  |  |  |  |

NOTE: Δ denotes blank.

## TABLE X-12 - RESULTS AFTER PASS 10

| Index | Triples | | | Contributing statement | | |
|---|---|---|---|---|---|---|
| 42 | F | = | $R_{40}$ | 1 | | |

| | Resulting statements | | | | | | |
|---|---|---|---|---|---|---|---|
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ► | Δ | Δ | Δ | Δ | Δ | Δ |
| 2 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 3 | $R_{42}$ | Δ | Δ | Δ | Δ | Δ | Δ |
| 4 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 5 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 6 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 7 | Δ | Δ | Δ | Δ | Δ | Δ | Δ |
| 8 | Δ | Δ | Δ | Δ | | | Δ |
| 9 | Δ | Δ | Δ | Δ | | | Δ |
| 10 | Δ | Δ | Δ | Δ | | | Δ |
| 11 | Δ | Δ | Δ | Δ | | | Δ |
| 12 | Δ | Δ | Δ | Δ | | | |
| 13 | Δ | Δ | Δ | Δ | | | |
| 14 | Δ | Δ | Δ | | | | |
| 15 | Δ | Δ | Δ | | | | |
| 16 | Δ | Δ | Δ | | | | |
| 17 | Δ | Δ | Δ | | | | |
| 18 | Δ | Δ | Δ | | | | |
| 19 | Δ | Δ | Δ | | | | |
| 20 | Δ | Δ | Δ | | | | |
| 21 | Δ | Δ | Δ | | | | |
| 22 | Δ | Δ | Δ | | | | |
| 23 | Δ | Δ | Δ | | | | |
| 24 | Δ | Δ | Δ | | | | |
| 25 | Δ | Δ | | | | | |
| 26 | Δ | Δ | | | | | |
| 27 | ◄ | Δ | | | | | |
| 28 | | Δ | | | | | |

NOTE: Δ denotes blank

## TABLE X-13 - RESULTS AFTER PASS 11

| Index | Triples | | | Contributing statement | | | |
|---|---|---|---|---|---|---|---|
| None | None | | | None | | | |

| | Resulting statements | | | | | | |
|---|---|---|---|---|---|---|---|
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | △ | △ | △ | △ | △ | △ | △ |
| 2 | △ | △ | △ | △ | △ | △ | △ |
| 3 | △ | △ | △ | △ | △ | △ | △ |
| 4 | △ | △ | △ | △ | △ | △ | △ |
| 5 | △ | △ | △ | △ | △ | △ | △ |
| 6 | △ | △ | △ | △ | △ | △ | △ |
| 7 | △ | △ | △ | △ | △ | △ | △ |
| 8 | △ | △ | △ | △ | | | △ |
| 9 | △ | △ | △ | △ | | | △ |
| 10 | △ | △ | △ | △ | | | △ |
| 11 | △ | △ | △ | △ | | | △ |
| 12 | △ | △ | △ | △ | | | |
| 13 | △ | △ | △ | △ | | | |
| 14 | △ | △ | △ | | | | |
| 15 | △ | △ | △ | | | | |
| 16 | △ | △ | △ | | | | |
| 17 | △ | △ | △ | | | | |
| 18 | △ | △ | △ | | | | |
| 19 | △ | △ | △ | | | | |
| 20 | △ | △ | △ | | | | |
| 21 | △ | △ | △ | | | | |
| 22 | △ | △ | △ | | | | |
| 23 | △ | △ | △ | | | | |
| 24 | △ | △ | △ | | | | |
| 25 | △ | △ | | | | | |
| 26 | △ | △ | | | | | |
| 27 | △ | △ | | | | | |
| 28 | | △ | | | | | |

NOTE: △ denotes blank.

## TABLE X-14 - TRIPLE SUMMARY

| Index | Triples | | | Contributing statement |
|---|---|---|---|---|
| 1 | A | * | X | 1 |
| 2 | O | - | S | 1 |
| 3 | O | - | B | 2 |
| 4 | B | * | B | 2 |
| 5 | 4 | * | A | 2 |
| 6 | 2 | * | A | 2 |
| 7 | A | + | E | 3 |
| 8 | E | * | P | 3 |
| 9 | O | .ABS. | X | 3 |
| 10 | A | + | B | 4 |
| 11 | D | * | E | 4 |
| 12 | F | + | G | 5 |
| 13 | H | * | I | 6 |
| 14 | U | .P. | V | 7 |
| 15 | B | + | $R_1$ | 1 |
| 16 | O | .ABS. | $R_2$ | 1 |
| 17 | $R_5$ | * | C | 2 |
| 18 | $R_8$ | / | $R_9$ | 3 |
| 19 | $R_{10}$ | + | C | 4 |
| 20 | U | * | $R_{12}$ | 5 |
| 21 | V | * | $R_{13}$ | 6 |
| 22 | R | .P. | $R_{16}$ | 1 |
| 23 | $R_4$ | - | $R_{17}$ | 2 |
| 24 | D | + | $R_{18}$ | 3 |
| 25 | $R_{19}$ | - | $R_{11}$ | 4 |
| 26 | O | .ABS. | $R_{14}$ | 7 |
| 27 | X | * | $R_{15}$ | 1 |
| 28 | 1 | * | $R_{25}$ | 4 |
| 29 | O | - | $R_{26}$ | 7 |
| 30 | C | + | $R_{27}$ | 1 |
| 31 | $R_{23}$ | .P. | ? | 2 |
| 32 | C | * | $R_{24}$ | 3 |
| 33 | W | * | $R_{29}$ | 7 |
| 34 | $R_3$ | + | $R_{31}$ | 2 |
| 35 | X | * | $R_{30}$ | 1 |
| 36 | $R_7$ | .P. | $R_{32}$ | 3 |
| 37 | D | + | $R_{35}$ | 1 |
| 38 | $R_{34}$ | / | $R_6$ | 2 |
| 39 | H | * | $R_{36}$ | 3 |
| 40 | $R_{37}$ | + | $R_{22}$ | 1 |
| 41 | O | * | $R_{38}$ | 2 |
| 42 | F | * | $R_{40}$ | 1 |

The compilation algorithm may generate triples involving variables not available at the time the triples are formed. For example, Pass 1 generated these triples:

| Index | | Triple | |
|-------|-----|-------|-----|
| 12 | F | + | G |
| 13 | H | * | I |
| 14 | U | . P. | V |

Now the resultant for Triple 12 [13] cannot be calculated until Statements 1 and 2 [3 and 4] are executed. Similarly, the resultant for Triple 14 cannot be calculated until Statements 5 and 6 are executed.

This "premature" generation of triples should pose no problem in the parallel processor, since Machine I (Appendix III) provides a "compute on availability" option. That is, if the quantity $A + B$ is to be computed, the machine will delay the computation until such time as $A$ and $B$ are available.

# APPENDIX XI - FURTHER NOTES ON PARALLEL COMPILATION

## 1. INTRODUCTION

The notion of parallel compilation is discussed and an algorithm for effecting it is presented in Appendix X. Subsequent attempts to implement this algorithm on a parallel processor (Appendices VI and XV) revealed the desirability of modifying it because implementation of the algorithm in its present form may require the initiation of an excessive number of parallel processor "tasks" (Appendixes XIV and XV) and lead to extremely cumbersome control programs.

In this appendix, three modifications of the parallel compilation algorithm are suggested. The first involves the translation of MAD[1,a] statements into reverse Polish notation[2] and testing for triple formation in parallel with input operations. In the second, the tests for triple formation are selectively applied as compilation progresses. In the third the form of the algorithm is changed.

Throughout this appendix, the class of MAD statements considered is restricted to replacement type statements involving nonsubscripted variables.

For a review of parallel compilation, see Appendix X.

## 2. PROBLEMS OF IMPLEMENTATION

The parallel compilation algorithm described in Appendix X specifies a sequence of passes in which the concurrent application of a set of tests to a list of replacement statements written in the MAD language results in all possible triple formations and/or statement simplifications. The algorithm, as presented in Figure X-1 of Appendix X, suggests that application of all the tests to all possible sequences of contiguous items

---

[a] Superior numbers in the text refer to items in the List of References, Item 5, Page 283.

taken 3, 4, and 5 at a time. Investigation, however, reveals that in a computer implementation of the algorithm such a procedure would be wasteful of machine capacity. As a case in point, consider the expression

$$F = A + B* . ABS. (C + D) . \tag{1}$$

On the first pass over (1), the algorithm would combine C, +, D into a triple, say $R_1$. Clearly, on the next pass it is futile to look at items preceding the sequence (, R1, ) in the hope of obtaining triple formations and/-or statement simplifications. Hence, to assign machine capacity to such testing is wasteful.

While this potentially wasteful testing is easily recognized, its remedy is not. Just how one might program a parallel processor to test those (and only those!) contiguous sequences of items where the specified conditions may obtain is a problem dealt with in Item 3, below.

Yet another problem incident to the implementation of the algorithm stems from the fact that the several tests, being of different complexity, require different amounts of time for machine execution.

This disparity of execution time leads to some rather severe programming problems associated with triple formation and/or statement simplification and maintenance of a valid item list. These programming problems should not be attributed entirely to the form of the parallel compilation algorithm; much of the difficulty of program construction is due to the fact that both the parallel processor and its associated programming language are radically new and there exists but little experience on which to draw in the construction of programs.

As a means of obviating the problems mentioned above, the following possibilities were considered: preliminary translation of input statements to reverse Polish notation, and innovations in the utilization of the parallel processor programming language. Both possibilities were investigated with fruitful results. An unexpected result of the investigation was the

development of a completely new form of the compilation algorithm. The results are detailed in Item 3.

### 3.  SUGGESTED MODIFICATIONS

#### a.  General

In this section, two methods of implementing the parallel compilation algorithm on the parallel processor are discussed. The first method involves the conversion of MAD statements into reverse Polish notation (RPN); the second involves programming innovations. Subsequently, a restructuring of the parallel compilation algorithm that offers greatly increased speed of execution is presented.

#### b.  Reverse Polish Notation (RPN)

Polish notation refers to a method of representing logical formulas developed by a Polish mathematician, J. Lukasiewicz.[3] The notation provides an unambiguous sequential specification of the order of execution of logical and arithmetic operations without the use of parentheses. The particular form of Polish notation used here is RPN. In RPN, operators are written after the operands on which they operate. A necessary condition for the use of RPN is that each operator be associated with a definite number of operands. For example, A + B would be written in RPN as AB+ where + would be understood to operate on the two operands immediately preceding it (A and B). Operands may in fact be the results of operations. For example, the expression A+B*C is written in RPN as ABC*+ where * operates on B and C, giving B*C, and the + operates on A and BC*, giving A+B*C. The ability of RPN to obviate the use of parentheses is seen by considering the expression (A + B)*C, which would be written as AB + C*.

Implementation of the parallel compilation algorithm may be facilitated by first translating MAD statements into RPN and then constructing triples from the RPN representation. The MAD statement (1):

-265-

$$F = A + B* .ABS. (C + D) ,$$

would appear in RPN as

$$FABCD + .ABS. * + = \qquad (2)$$

which, for purposes of compilation, would be interpreted as

| | | |
|---|---|---|
| $R_1$ | CD+ | C + D |
| $R_2$ | $R_1$ .ABS. | 0 .ABS. $R_1$ |
| $R_3$ | $BR_2$* | B * $R_2$ |
| $R_4$ | $AR_3$+ | A + $R_3$ |
| $R_5$ | $FR_4$= | F = $R_4$ |

which is read as

$$F = R_4$$
$$= A + R_3$$
$$= A + B*R_2$$
$$= A + B* .ABS. R_1$$
$$= A + B* .ABS. (C + D)$$

which is just (1).

Figure XI-1 gives a flow chart that specifies a method of translating MAD statements into RPN. The method depends upon the specification of precedences for operators and the use of a stack. Table XI-1 gives the required precedences. Note that these precedences differ slightly from those of Table I in Appendix X in that it includes operators allowing the translation of statements involving logical operations.
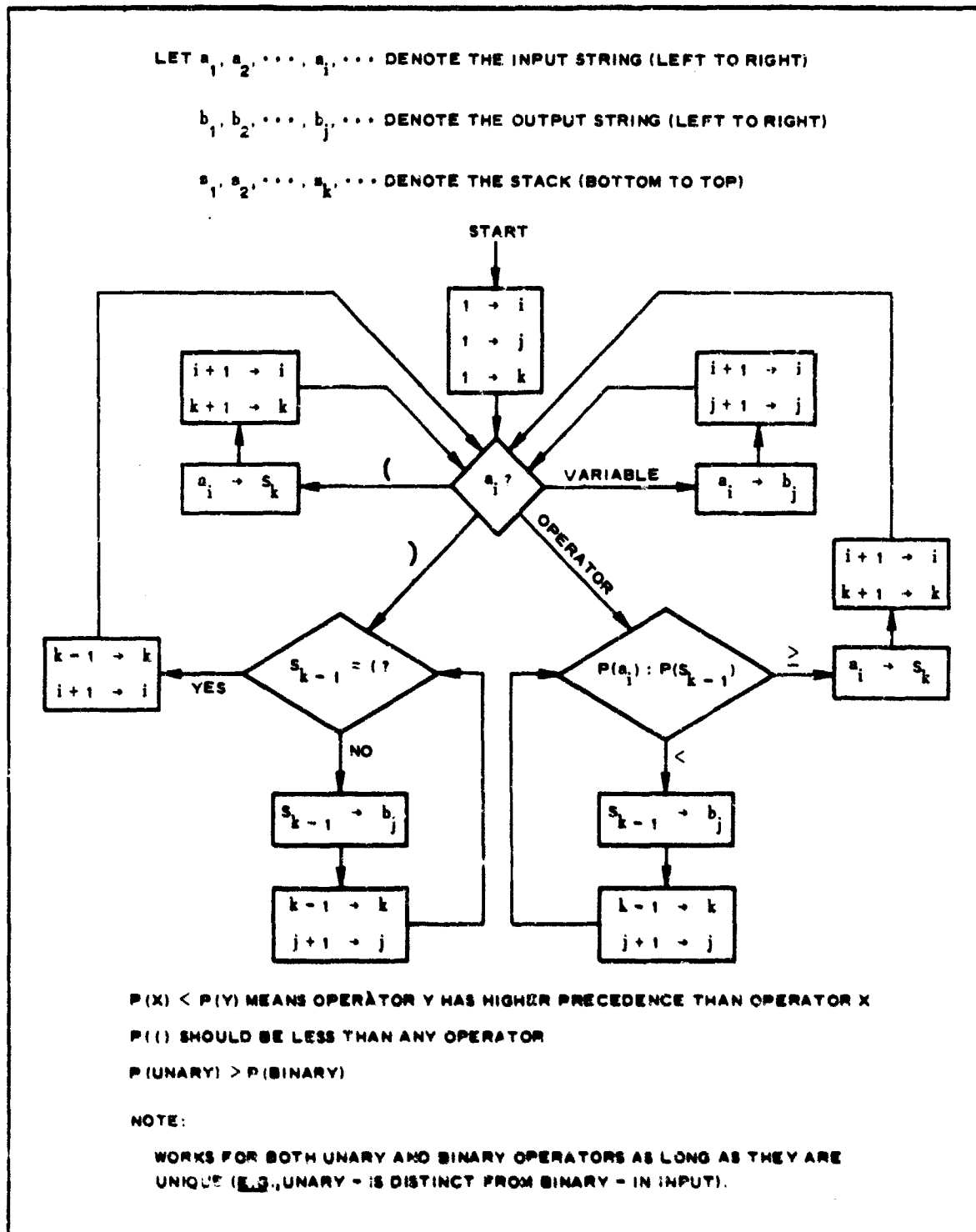
Figure XI-1 - Translation from MAD to Reverse Polish Notation

## TABLE XI-1 - PRECEDENCE HIERARCHY FOR

## RPN TRANSLATION

| Operator | Description | Precedence |
|----------|-------------|------------|
| .ABS., $-_u$ | Absolute value, unary minus | Highest |
| .P. | Exponentiation | |
| *, / | Multiplication, division | |
| +, - | Addition, subtraction | |
| $<$, $\leq$, $>$, $\geq$, $\neq$, $=$ | Relations (with usual interpretations) | |
| $\sqcap$ | Not | |
| $\wedge$ | And | |
| $\vee$ | Or | |
| := | Equals (substitution)* | |
| $\vdash$, $\dashv$, (, ) | Begin statement, and statement, open parenthesis, close parenthesis | Lowest |

*In the accompanying discussion only := occurs and is written as =

The translation proceeds basically as follows. Items are taken, left to right, from the input string. If an item is a variable, it is added, in a left-to-right fashion, to the output string. If an item is an operator, its precedence is compared with the precedence of the topmost item in the stack. If the precedence of the input item is greater than or equal to the precedence of the stack item, the input item is added to the top of the stack; otherwise, the input item "sinks" down the stack until it encounters a stack item whose precedence is no greater than that of the input item. As an input item sinks in the stack, those stack items whose precedences exceed that of the input item are removed from the top of the stack and added to the output string.

Parentheses require special comment. An open parenthesis, when encountered, is placed on the top of the stack. A close parenthesis, when encountered, sinks in the stack until a stacked open-parenthesis is encountered at which point both are removed from further consideration. Those stack items past which a close parenthesis sinks are added to the output string.

Certain comments are also in order regarding relative precedences of unary and binary operators involved in the translation process. It is necessary, first of all, for unary operators to have higher precedence than binary operators. Consider the following cases. The string a. P. b. P. c may be interpreted either as $a^{bc}$ or $a^{b^c}$, the choice being determined by the direction of scan (left to right or right to left) of the input string and the placement of the equality sign on the paths out of the precedence test box for operators (see Figure XI-1). But the string a + B* . ABS. C admits of only one interpretation, namely a + (b*( .ABS. (c))). With the precedence of unary operators greater than those of binary operators, the expression a + b* .ABS. C would translate, correctly, into RPN as abc .ABS. *+. However, if the unary operation had lower precedence, the translation would produce, incorrectly, abc* + .ABS. which implies .ABS. (a + b*c).

Further, all unary operators should be of equal precedence. Suppose one had to translate an input string . . . $B_1 U_1 U_2$ . . . $U_n V B_2$ . . ., where the B's represent binary operators, the U's unary operators, and V a variable. (Many unary operators are possible; for example, $-_u$, .ABS., sin, log, etc.) Correct translation requires that the output string appear in RPN as . . . $V U_n U_{n-1}$ . . . $U_1$ . . . which, in turn, requires that the unary operators have equal precedence.

The process of parallel compilation utilizes RPN in the following fashion. As MAD statements are read into the parallel processor, they are converted into RPN. As soon as an operator is inserted into the RPN string, the parallel processor begins the formation of the

-269-

corresponding triple while continuing the read-in process. This procedure allows compilation to proceed in parallel with input.

The use of RPN in the compilation procedure, as outlined above, greatly simplifies the task of the programmer in implementing the compilation algorithm on a parallel processor. However, the use of RPN in the implementation of the compilation algorithm seems unavoidably to result in a parallel processor machine program that is rather slow when applied to a single statement. The slowness of a program using RPN is due mainly to the sequential nature of the translation process and the well-known inefficiency accompanying the constraint of the parallel processor to a single sequential mode of operation. Although the effects of this can be mitigated by the concurrent compilation of many statements on the parallel processor, further efforts were made to discover a simple, easily programmed method of implementing the compiler algorithm that would be free of slowness due to inherent sequential characteristics. Some results of these efforts follow.

c.  Programming Innovations

As pointed out in Item 2 above, initial attempts to implement the parallel compilation algorithm met with severe programming problems which in turn led to the use of RPN in the compilation process. But the use of RPN was not wholly satisfactory. Hence, renewed efforts were made to construct an efficient compiler program free of the sequential limitations inherent in the use of RPN. The result was an implementation whose flow diagram appears in Figure XI-2.

The implementation of the compilation algorithm as given in this illustration proceeds as follows. Reading left to right, the $i^{th}$ item in a MAD replacement statement is denoted by $L_i$; with each $L_i$ is associated a number, $K_i$, which may be 1, 2, or 3. To each $L_i$ is assigned a parallel processor task (the tasks proceed in parallel) that determines whether or not $L_i$ may be used in triple formation or statement

$L_i$ IS $i$

$L_i$ IS $\vdash$

$L_{i+2}$? — OTHER — $2 \rightarrow K_i$

$L_i \in \{\vdash, )\}$

$L_j \in \{\vdash, \sim_b\}$

$L_{i+1}$? — OTHER — $2 \rightarrow K_i$

$1 \rightarrow K_i$

$\sim_u (, .ABS.$

$L_{i+1}$?  — V

$L_{i-}$

$L_{i+2}$? — OTHER

READ AND ERASE $K_{i+2}$

$L_{i+2}$?

$3 \rightarrow K_i$ — <

$P(L_i) : P(L_{i+2})$?

$L_i$

ERASE $L_i, L_{i+1}, L_{i+2}$

$\geq$
$=$

$1 \rightarrow K_i$

$)$

READ AND ERASE $K_{i+3}$

READ AND ERASE $K_{i-1}$

READ AND ERASE $K_{i+2}$

TREAT $L_{i-2}$ AS $\sim_u$ — OTHER — $L_{i-3}$? — $L_{i-2}$?

$), V$

TREAT $L_{i-2}$ AS $\sim_b$

OTHER

WAIT FOR ALL THREE

$P(L_{i-2}) : P(L_i)$?

ERASE $L_i, L_{i+2}$ — $K_{i+3}$?

$< $  $\geq$
$=$

$2 \rightarrow K_i$    $3 \rightarrow K_i$

1    3

START $i+3$

$2 \rightarrow K_{i+3}$

$K_{i-1}$?

2    3

START $i-1$

$1 \rightarrow K_{i-1}$

V = VARIABLE

$\bullet$ = STOP TESTING

$P(x) : P(y)$? = WHAT IS THE RELATION OF $P(x)$ TO $P(y)$

$\sim_u$ = UNARY MINUS

$\sim_b$ = BINARY MINUS

$\sim$ = UNDETERMINED MINUS

START $i-2$ — 2 — $K_{i-2}$?

3

$1 \rightarrow K_{i-2}$

A

Figure XI-2 - Flow Chart for Parallel Compilation

-271-

simplification, tasks assigned to variables or resultants of triples are suspended; tasks assigned to operators form triples or statement simplifications if test conditions are met. If test conditions fail to be met, the task detects whether failure is due to inappropriate items on the left side, the right side, or both sides. This information is stored in $K_i$ and used to re-initiate testing whenever changes are made in items immediately to the left, right, or both sides. $K_i = 1, 2,$ or 3 indicates blocking on the left, right, or both, respectively.

NOTE: In Figure XI-2, the precedences of Table XI-1 are used. $L_{i \pm j}$ denotes the item j places to the right (+) or left (-) of $L_i$; that is, "blanks" generated by triple formation and/or statement simplification are ignored.

The implementation of the parallel compiler algorithm specified by Figure XI-2 offers a solution to a problem posed in Item 2, namely: How does one program a parallel processor to test those and only those contiguous sequences of items where specified test conditions may obtain. This, of course. allows economical use of machine capacity in that all futile testing is avoided and tasks are assigned only to those items where triple formation and/or statement simplification is occuring. Further, the control features given in Figure XI-2 (for example, the "wait for both" boxes) allow the maintenance of a valid and unambiguous item list throughout compilation.

Although the method of Figure XI-2 appears to offer an efficient, easily programmed implementation of the parallel compilation algorithm, preliminary timing estimates indicate that it is not significantly faster than the RPN method.

### d. Restructuring of the Algorithm

An examination of the compiler algorithm reviewed in Appendix X reveals that total parallelism of compilation has not been achieved. The reason is as follows: Although the algorithm specifies on each pass

the concurrent execution of all possible triple formations and/or
statement simplifications, the procedure is limited in that it exam-
ines only contiguous sequences of items taken 3, 4, or 5 at a time.
Hence, several sequential passes of the algorithm are necessary to
complete compilation. This limitation is also present in the modifi-
cations described above.

The question arises as to whether or not an algorithm can be devel-
oped that will concurrently examine each item of a list in terms of
all other items with which it may ultimately be associated in the com-
pilation process, and specify triple formation and/or statement sim-
plification in a fashion that achieves optimal compilation speeds.
This question is now considered.

Consider the MAD statement,

$$Z = (A + B)*C + D*((E*F + G)*H + I) , \qquad (3)$$

and the precedence assignment

| Symbol | Precedence | |
|--------|------------|---|
| Variable | 3 + 4N | |
| * | 2 + 4N | (4) |
| + | 1 + 4N | |
| = | 0 + 4N | |

where N denotes the number of parentheses sets enclosing the sym-
bol.

Using the effective precedence due to parentheses inclusion, (3) can
be written in a list as

| Symbol | Precedence | Symbol | Precedence | |
|--------|------------|--------|------------|---|
| Z | 3 | + | 5 | |
| = | 0 | B | 7 | (5) |
| A | 7 | * | 2 | |

| Symbol | Precedence | Symbol | Precedence | |
|--------|------------|--------|------------|----|
| C | 3 | + | 9 | |
| + | 1 | G | 11 | |
| D | 3 | * | 6 | (5) |
| * | 2 | H | 7 | |
| E | 11 | + | 5 | |
| * | 10 | I | 7 | |
| F | 11 | | | |

The list (5) may be interpreted in graphical form as shown in Figure XI-3. It will be noted in this graph how precedence modification, due to parentheses inclusion, separates groups of symbols on the basis of parenthetical grouping and obviates the need of further retention of parentheses. The beginning and end points of the statement are arbitrarily assigned a precedence of $-\infty$.
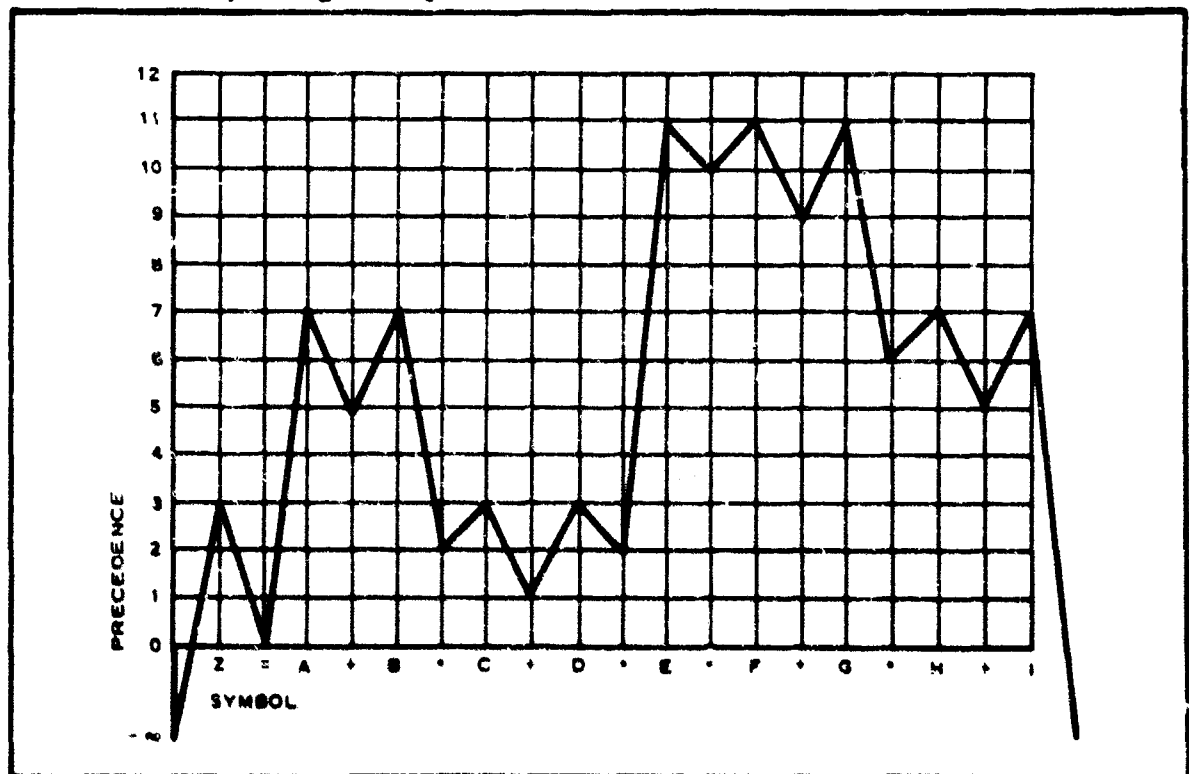


Figure XI-3 - Graphical Interpretation of List (5)

In the process of compilation, each triple is formed from two ordered variables (or resultants) and a binary operator (one variable or resultant and a unary operator). Consider a single variable that is preceded, and followed, by binary operators. In triple formation, this variable will be included in the triple corresponding to whichever of the two binary operators is of higher precedence. For example, from the statement (3) select $(A + B)*C$ which is stored in the list (5) as

| Symbol | Precedence |
|--------|------------|
| A | 7 |
| + | 5 |
| B | 7 |
| * | 2 |
| C | 3 |

Select the variable B that is preceded by + and followed by *. Because of parentheses inclusion, the effective precedence of + is 5 which is greater than 2, the precedence of *, and thus B is to be used in the triple corresponding to +. B will be used in the right side of the triple corresponding to + since + is on the left side of B.

As shown in Figure XI-3, B is at a peak on the graph as are all variables. To find the operator in whose triple a variable will be included is quite simple. One simply "looks down the slopes" to find the "nearest" (numerically greatest) operator, with which the variable is then associated. In the event that a variable has operators of equal precedence on either side, the right operator will be considered as having greater precedence. This convention conforms to the required interpretation of precedences in a concatenation of unary operators.

Now, since each operator will generate a triple, consideration must be given to the placement of the corresponding resultant in other triples. An examination of Figure XI-3 will quickly suggest how

resultants can be combined into triples. Each operator corresponds
to a "valley" (perhaps "plateau" in the case of concatenated unary op-
erators). But each operator also represents a triple and correspond-
ing resultant that must be treated as a variable. Hence, for each
operator one must search the graph, both to the right and the left,
until on each side an operator of lesser precedence is encountered.
The resultant is then associated with the triple corresponding the op-
erator of higher precedence (rightmost operator in the case of equality).

Basically, then, triple formation proceeds in a leveling process that
consists of combining variable "peaks" of a graph such as that given
in Figure XI-3 into "valley" triples whose resultants are then treated
as variables and the process iterated. Parallelism is injected into
the procedure by concurrently associating each variable and resultant
with the triple in which it will ultimately be located.

The method of compiling MAD statements into triples outlined above
involves searches for items of lesser precedence both to the right and
left of a given item from a list such as (5). These searches can be
accomplished more easily if such a list is stored as a part of an ex-
panded list defined as follows:

1.  Let n symbols, such as those in the list (5), be
    indexed $1, 2, \ldots, n$

2.  Let p be the integer such that $2^{p-1} < n + 2 \leq$
    $2^p$. Construct a list of items indexed

    $$2, 3, 4, \ldots, 2^p, 2^p + 1, 2^p + 2, \ldots,$$

    $$2^p + n, 2^p + n + 1 \qquad (6)$$

    where n denotes the number of symbols from a
    list such as (5) and $2^p + i$ denotes the index for
    symbol i from a list such as (5). The other in-
    dices of (6) denote dummy items that will be

assigned a precedence defined below. These dummy items will be of value in treeing the searches for items to the left or right of a given item that are of lesser precedence. Figures XI-4 and XI-5 present methods for the requisite searches.

3. Using the precedences given in (7), or in some similar but more comprehensive list, denote by $M(j)$ the precedence of the $j^{th}$ item of (6). For $2^p + 1 \leqq j \leqq 2^p + n$, $M(j) = M(2^p + i)$, the precedence of the $i^{th}$ symbol from a list such as (7).

4. Let $M(2^p) = M(2^p + n + 1) = -\infty$. That is, drive the endpoints of a graph such as that given in Figure 4 to minus infinity.

5. For items of the list (8) indexed j, $2 \leqq j < 2^p$, define precedence as follows:

   a. If $M(2j)$ and $M(2j + 1)$ are defined, let $M(j) = \min [M(2j), M(2j + 1)]$

   b. If only $M(2j)$ is defined, let $M(j) = M(2j)$

   c. If neither $M(2j)$ or $M(2j + 1)$ is defined, then $M(j)$ is undefined

   Undefined items will not affect the search procedures specified in Figures 7 and 8.

Table XI-2 illustrates the compilation procedure described above. The statement (3) in the list form (5) is used as an example.

An example of precedence determination for an expanded list such as (6) is given in Table XI-3. Again the statement (3) is used as an example. The number of symbols in statement (3), excluding parentheses, is n = 19, hence for p such that $2^{p-1} < n + 2 \leqq 2^p$, p = 5. This accounts for the expanded list index running from 2 through 52 in Table XI-3.

-278-

Figure XI-4 - Subroutine for Finding $L_j$

$R_j$ IS THE INDEX OF THE LEFT-MOST SYMBOL TO THE RIGHT OF
SYMBOL j WITH LESS PRECEDENCE THAN THAT OF SYMBOL j.

FOR ALL j'S, $1 \leq j \leq n$
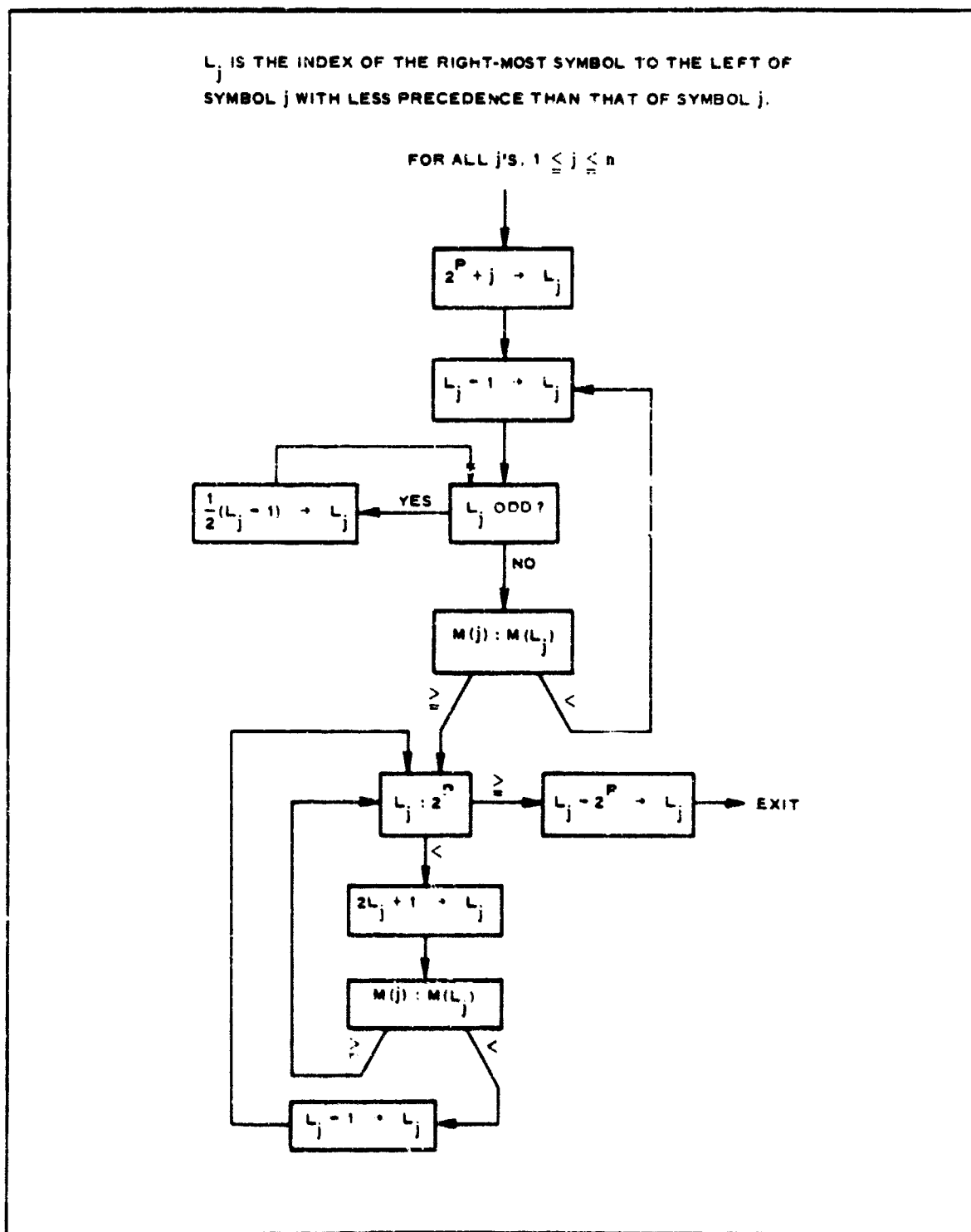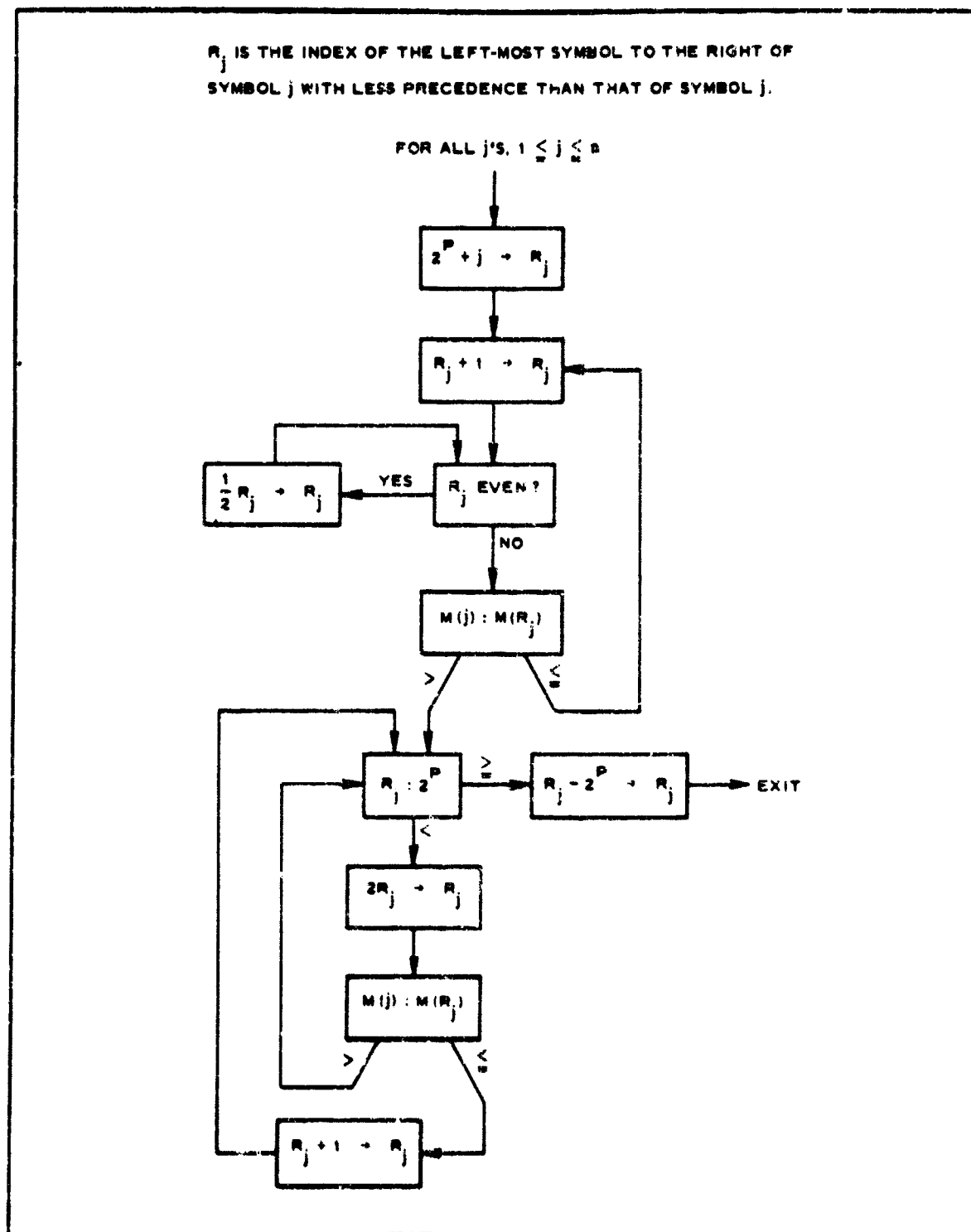
Figure XI-4 - Subroutine for Finding $R_j$

## TABLE XI-2 - EXAMPLE OF THE COMPILATION

## PROCEDURE FOR STATEMENT (3)

| Index j | Symbol j | Precedence of symbol j | Lj | Rj | Higher precedence item of Lj and Rj | Triple |
|---|---|---|---|---|---|---|
| 1 | Z | 3 | 0 | 2 | 2 | Z |
| 2 | = | 0 | 0 | 20 | 20 | ①=⑧ |
| 3 | A | 7 | 2 | 4 | 4 | A |
| 4 | + | 5 | 2 | 6 | 6 | ③+⑤ |
| 5 | B | 7 | 4 | 6 | 4 | B |
| 6 | * | 2 | 2 | 8 | 8 | ④*⑦ |
| 7 | C | 3 | 6 | 8 | 6 | C |
| 8 | + | 1 | 2 | 20 | 2 | ⑥+⑩ |
| 9 | D | 3 | 8 | 10 | 10 | D |
| 10 | * | 2 | 8 | 20 | 8 | ⑨*⑱ |
| 11 | E | 11 | 10 | 12 | 12 | E |
| 12 | * | 10 | 10 | 14 | 14 | ⑪*⑬ |
| 13 | F | 11 | 12 | 14 | 12 | F |
| 14 | + | 9 | 10 | 16 | 16 | ⑫+⑮ |
| 15 | G | 11 | 14 | 16 | 14 | G |
| 16 | * | 6 | 10 | 18 | 18 | ⑭*⑰ |
| 17 | H | 7 | 16 | 18 | 16 | H |
| 18 | + | 5 | 10 | 20 | 10 | ⑯+⑲ |
| 19 | I | 7 | 18 | 20 | 18 | I |

NOTE: Lj[Rj] denotes the rightmost [leftmost] symbol to the left [right] of symbol j having precedence less than that of symbol j.

## TABLE XI-3 - EXAMPLE OF LIST EXPANSION AND PRECEDENCE DETERMINATION FOR STATEMENT (3)

**Statement**

$$Z = (A + B)*C + D*((E*F + G)*H + I) \qquad (3)$$

**Precedence table**

| Symbol | Precedence | |
|--------|------------|--|
| Variable | 3 | Add 4 for every set of |
| * | 2 | parentheses enclosing a |
| + | 1 | symbol |
| = | 0 | |

$n = 19, \; 2^{p-1} < n + 2 \leqq 2^p \Rightarrow p = 5$

| i | M(i) | i | M(i) | i | M(i) |
|---|------|---|------|---|------|
| 2 | $-\infty$ | 19 | 3 | 36 | 5 + |
| 3 | $-\infty$ | 20 | 1 | 37 | 7 B) |
| 4 | $-\infty$ | 21 | 2 | 38 | 2 * |
| 5 | 1 | 22 | 10 | 39 | 3 C |
| 6 | $-\infty$ | 23 | 9 | 40 | 1 + |
| 7 | | 24 | 6 | 41 | 3 D |
| 8 | $-\infty$ | 25 | 5 | 42 | 2 * |
| 9 | 2 | 26 | $-\infty$ | 43 | 11 ((E |
| 10 | 1 | 27 | | 44 | 10 * |
| 11 | 9 | 28 | | 45 | 11 F |
| 12 | 5 | 29 | | 46 | 9 + |
| 13 | $-\infty$ | 30 | | 47 | 11 G) |
| 14 | | 31 | | 48 | 6 * |
| 15 | | 32 | $-\infty$ | 49 | 7 H |
| 16 | $-\infty$ | 33 | 3 Z | 50 | 5 . |
| 17 | 0 | 34 | 0 = | 51 | 7 I) |
| 18 | 5 | 35 | 7 (A | 52 | $-\infty$ |

Figures XI-4 and XI-5, respectively, present flow charts of search procedures for finding $L_j$ and $R_j$, where $L_j[R_j]$ denotes the rightmost [leftmost] symbol to the left [right] of symbol j having precedence less than that of symbol j. These search procedures can be executed concurrently in approximately $2 \ln_2 n$ steps where n is the number of symbols (less parentheses) from a statement such as (3).

## 4. CONCLUSIONS

In this appendix, three modifications of the parallel compilation algorithm have been presented. The first involved preliminary translation of replacement statements into reverse Polish notation; the second involved innovations in the use of the parallel processor programming language; the third specified a restructuring of the algorithm.

The first two modifications were easily programmable but did not result in significant speed advantages. The restructuring offered by the third modification appears to provide a maximal utilization of parallelism inherent in the compilation process. The restructured compilation algorithm has not yet been programmed for a parallel processor nor has it been subjected to a detailed review. It is recommended that further study of the restructured parallel compilation algorithm be carried out.

## 5. LIST OF REFERENCES

1.  University of Michigan Computing Center: _Michigan Algorithm Decoder_. Ann Arbor, Mich., June 1963.

2.  Wagner, P. (editor): _Introduction to System Programming_. New York, N. Y., Academic Press, 1964.

3.  Arden, B., Galler, B., and Graham, R.: "An Algorithm for Translating Boolean Expressions." _Journal of the ACM_, April 1962.

APPENDIX XII - PROGRAMMING OF THE SEQUENTIAL

COMPILATION ALROGITHM FOR THE IBM 7090


1. INTRODUCTION

This work was performed under Contract AF30(602)-3550, Advanced Com-
puter Organization Study. A sequential algorithm for compiling substitution
expressions was written so that a comparative analysis to a parallel ma-
chine could be made. The IBM 7090 computer was chosen as the sequential
computer on which to make the comparisons.[a]

Reference was first made to a paper by Arden, Galler, and Graham.[b] An
extensive analysis was made of the compilation of a general complex sub-
stitution expression, and further investigation led to a general derivation
of a timing equation for compiling simplified expressions.

The comparative analysis of sequential versus parallel compilation is made
in Appendix XIII.


2. DESCRIPTION OF ALGORITHM

a. General

In this IBM 7090 compiler algorithm, reasonable assumptions have
been made as to what will be the format of the input string or substi-
tution expressions. Due to the hierarchy of the operators, the oper-
ands are considered as having a zero level of hierarchy. The operators

---

[a]IBM Reference Manual, 7090 Data Processing System. Poughkeepsie, N. Y.,
International Business Machines Corporation, August 1961.

[b]Arden, B. W.; Galler, B. A.; and Graham, R. M.: An Algorithm for Trans-
lating Boolean Expressions. Ann Arbor, Mich.. University of Michigan, Oc-
tober 1961.

listed below constitute an unalterable basic set whose meaning (semantic content) is used in the decomposition of expressions. Boolean expressions will not be considered here but it is easily seen that to include them one would merely extend the limits of the algorithm. All arithmetic operators except the exponential will be generated into an object program in single precision arithmetic. The overall program package will require input/output routines and an exponential subroutine (EXP) that is available in the MAD compiler. The symbol "-" is used in statements to indicate both the unary (one operand) and the binary (two operands) operator; the context indicates which is intended.

Certain arithmetic operations must be compiled first in order to execute the object program correctly. It is for this reason that a certain level of hierarchy is assigned to each of the input string of items. In this algorithm, the chosen hierarchy (or precedence) is as shown in Table XII-1.

## TABLE XII-1 - HIERARCHY OF INPUT ITEMS

| Item | Definition | Precedence | |
|------|------------|------------|---|
| . ABS. | Absolute Value | 12 | |
| -u | Unary minus | 11 | Operators |
| ↑ | Exponentiation | 10 | |
| * | Multiplication | 9 | |
| / | Division | 8 | |
| - | Binary minus | 7 | |
| + | Plus | 6 | Operators |
| = | Substitution | 5 | |
| ( | Left parenthesis | 4 | |
| ) | Right parenthesis | 3 | |
| ⊢ | Left terminator | 2 | |
| ⊣ | Right terminator | 1 | |
| V | Constant or variable | 0 | Operand |

If the input items used in an expression are compiled in accordance with this hierarchy, there will result an object program in 7090 code that when executed will accomplish the desired arithmetic operations. Through out the discussion, the following symbols are used:

$PREC(\delta_j)$ = Precedence of current operator

$PREC(\delta_{j-1})$ = Precedence of previous operator

$S_\beta$ = Current referenced input string item

$L_\alpha$ = Current referenced intermediate string item

$\beta$ = Input string index

$\alpha$ = Intermediate string index

$R_i$ = Current triple resultant pointer

$M_{i1}$ = First generated instruction of a triple

$M_{i2}$ = Second generated instruction of a triple

$M_{i3}$ = Third generated instruction of a triple

$M_{i4}$ = Fourth generated instruction of a triple

The compiler discussed here involves an input pass that will assign the constant/variable machine locations and the correct hierarchy to the input string. Once the input string is in this form the various items are placed on a list (SLIST). The major function of the algorithm involves a single scanning of the expression from right to left (up the SLIST) and retaining operands, operations, relations, etc. on an intermediate list I (LLIST) until an operation or relation $(\delta_j)$ occurs which is of lower precedence then the immediately preceding operation on the list $(\delta_{j-1})$.

When such an operation or relation as $\delta_j$ is encountered, $\delta_{j-1}$ is compiled. Except for the case of exponentiation the compilation consists of creating object coding of three instructions ($M_{i1}$, $M_{i2}$, and $M_{i3}$).

The three instructions created for a + b are $M_{i1}$ = CLA a, $M_{i2}$ = FAD b, and $M_{i3}$ = $STOR_i$, where $R_i$ is the current location of the result of performing the operation a + b which will be called a triple for this discussion.

b. **Input String Discussion**

The input routine must be capable of reading a statement into the 7090 and assigning relative addresses to the various constants and variables (that is, count the different operands). If a variable or constant (operand) is detected, the sign bit of the machine word referencing its location (bits 21 to 35) is made one (bit 0) and the decrement (bits 3-17) is zero. An operand format is sketched below.

| 1 | (ZERO) $\phi$ | | RELATIVE LOCATION |
|---|---|---|---|
| S 1, 2, 3 | | 17 21 | 35 |
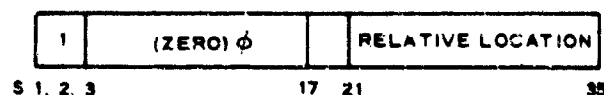
To execute the various arithmetic operations in the proper sequence, a certain level of hierarchy is assigned to each operator (see Page 284). In the case of operators, only the hierarchy mentioned on Page 284 is necessary to generate the correct object code and it must be contained in the decrement (bits 3 to 17) field of each item word. An operator format is sketched below.

| 0 | | $\delta$ | | (ZERO) $\phi$ |
|---|---|---|---|---|
| S 1, 2, 3 | | 17 | 21 | 35 |
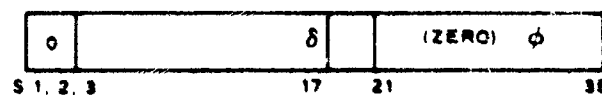
For example, should there be a complex expression such as

$$Z = (A*B + C*D) * (E\uparrow F + .ABS. 1) + (J\uparrow K - L/M)/(-P + Q*R) ,$$

the input routine would be expected to produce the array shown in Figure XII-1 (see Appendixes I and III).

c. **Input Data Discussion**

After the object program has been generated, the DATIN subroutine

INPUT STRING
(OCTAL)

CONSTANT, VARIABLE, AND
RESULTANT POOL

| SLiST S123 | DECREMENT 17 | 21 | ADDRESS 35 | | RESERVED FOR LEFT TERMINATOR POOL | 0 | 00000 | 0 | 00000 | (ZERO) |
|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 00000 | 0 | 00001 | Z | | | | | | Z |
| 0 | 00005 | 0 | 00000 | = | | | | | | A |
| 0 | 00004 | 0 | 00000 | ( | | | | | | B |
| 4 | 00000 | 0 | 00002 | A | | | | | | C |
| 0 | 00011 | 0 | 00000 | . | | | | | | D |
| 4 | 00000 | 0 | 00003 | B | | | | | | E |
| 0 | 00006 | 0 | 00000 | + | | | | | | F |
| 4 | 00000 | 0 | 00004 | C | | | | | | I |
| 0 | 00011 | 0 | 00000 | . | | | | | | J |
| 4 | 00000 | 0 | 00005 | D | | | | | | K |
| 0 | 00003 | 0 | 00000 | ) | | | | | | L |
| 0 | 00011 | 0 | 00000 | . | | | | | | M |
| 0 | 00004 | 0 | 00000 | ( | | | | | | P |
| 4 | 00000 | 0 | 00006 | E | | | | | | Q |
| 0 | 00013 | 0 | 00000 | ↑ | | | | | | R |
| 4 | 00000 | 0 | 00007 | F | | | | | | |
| 0 | 00006 | 0 | 00000 | + | | | | | | |
| 0 | 00014 | 0 | 00000 | .ABS. | | | | | | |
| 4 | 00000 | 0 | 00010 | | | | | | | |
| 0 | 00003 | 0 | 00000 | ) | | | | | | |
| 0 | 00006 | 0 | 00000 | + | | | | | | |
| 0 | 00004 | 0 | 00000 | ( | | | | | | |
| 4 | 00000 | 0 | 00011 | J | | | | | | |
| 0 | 00013 | 0 | 00000 | ↑ | | | | | | |
| 4 | 00000 | 0 | 00012 | K | | | | | | |
| 0 | 00007 | 0 | 00000 | - | | | | | | |
| 4 | 00000 | 0 | 00013 | L | | | | | | |
| 0 | 00010 | 0 | 00000 | / | | | | | | |
| 4 | 00000 | 0 | 00014 | M | | | | | | |
| 0 | 00003 | 0 | 00000 | ) | | | | | | |
| 0 | 00010 | 0 | 00000 | | | | | | | |
| 0 | 00004 | 0 | 00000 | ( | | | | | | |
| 0 | 00007 | 0 | 00000 | - | | | | | | |
| 4 | 00000 | 0 | 00015 | P | | | | | | |
| 0 | 00006 | 0 | 00000 | . | | | | | | |
| 4 | 00000 | 0 | 00016 | Q | | | | | | |
| 0 | 00011 | 0 | 00000 | . | | | | | | |
| 4 | 00000 | 0 | 00017 | R | | | | | | |
| 0 | 00003 | 0 | 00000 | ) | | | | | | |

LOCATIONS ASSIGNED DURING THE
INPUT ROUTINE AND FILLED WITH
FLOATING POINT VALUES BY THE
DATIN ROUTINE

Figure XII-1 - Format of an Input String of Items

must be capable of storing floating point values in the memory locations selected by the INPUT subroutine. These values will start at POOL+1, etc. The first location POOL is zero. If the input data are in integer form, provision must be made to convert it to floating point numbers.

d. Flow Diagram General Description

In addition to the terminology used in the general discussion of this section, the following additional symbology pertains to the flow diagrams:

> → means "is transferred to"
> $R_i + 1 \rightarrow R_i$ means "-$R_i$ is increased by one"

The flow diagram in Figure XII-2 is by no means complete from a systems standpoint as mentioned above. The operational system will require input/output and, if desired, object program listing routines.

e. Subroutine Descriptions

(1) Absolute Value

The absolute value may be detected by (1) the initial scan, or (2) if preceeded by 1-triple. If a twelve is detected in the decrement field of an item on the input string (SLIST) during the initial scan, the subroutine SCOPEM + 2 is entered. This subroutine will produce three instructions: (1) CLA POOL, which makes the AC register equal to zero when executed; (2) FAM $L_{a-1}$, which adds to the AC register the magnitude of the contents of the previous item on the L list, providing it is not an operator; and (3) STO $R_i$, which stores the final result of the operation, .ABS. $L_{a-1}$.

The location of the result after executing these instructions is put on the L list to keep account of the intermediate steps. The result pointer is increased by one and the program returns to the start to

START

(AC)
$s_\beta = ?$

VARIABLE OR CONSTANTS

"+"

"("

OTHERS

START

$s_\beta \rightarrow L_\alpha$

$\alpha + 1 \rightarrow \alpha$

$\beta - 1 \rightarrow \beta$

START

IS PREC $\delta$ < PREC $\delta_{j-1}$

NO → $\vdash R_f \dashv$ CASE LIST AND EXECUTE

STOP

YES

$\delta_j \rightarrow$ TEMP STORE

$CLA - L_{\alpha-3} \rightarrow M_{i1}$

$STO - L_{\alpha-1} \rightarrow M_{i2}$

$NOP - R_{f+1} \rightarrow M_{i3}$

$R_f \rightarrow L_{\alpha-3}$

$\alpha - 2 \rightarrow \alpha$

$R_i + 1 \rightarrow R_i$

$\delta_j \rightarrow AC$

+

IS PREC $\delta_j$ < PREC $\delta_{j-1}$

NO → $(R_i)$ CASE

$L_{\alpha-1} \rightarrow L_{\alpha-2}$

$\alpha - 1 \rightarrow \alpha$

$\beta - 1 \rightarrow \beta$

START

YES

$\delta_j \rightarrow$ TEMP STORE

$CLA - L_{\alpha-1} \rightarrow M_{i1}$

$OP - L_{\alpha-3} \rightarrow M_{i2}$

$STO - R_i \rightarrow M_{i3}$

$R_f \rightarrow L_{\alpha-3}$

(FORM TRIPLE)

$R_i + 1 \rightarrow R_i$

$\alpha - 2 \rightarrow \alpha$

$\delta_j \rightarrow AC$

(

START

$AC \rightarrow L$

$\alpha + 1 \rightarrow \alpha$

$\beta - 1 \rightarrow$

IS PREC $\delta$ < PREC $\delta_{j-1}$

NO

YES

$\delta_i \rightarrow$ TEMP STORE

$CLA - L_{\alpha-1} \rightarrow M_{i1}$

$OP - L_{\alpha-3} \rightarrow M_{i2}$

$STO - R_i \rightarrow M_{i3}$

$R_i \rightarrow L_{\alpha-3}$

$R_i + 1 \rightarrow R_i$

$\alpha - 2 \rightarrow \alpha$

$\delta_j \rightarrow AC$

OTHERS

A

A

START

$(AC)$
$s_\beta = ?$

.ABS.

OTHERS

"$-$"

OTHERS

"$\uparrow$"

IS $s_{\beta - 1}$
AN OPERATOR
?

NO

IS $s_{\beta - 1}$
ABS?

YES

$(R_i)$ CASE

$L_{a-1} \rightarrow L_{a-2}$

$a - 1 \rightarrow a$

$\beta - 1 \rightarrow \beta$

NO

START

$AC \rightarrow L_a$

$a + 1 \rightarrow a$

$\beta - 1 \rightarrow \beta$

YES

NO

START

IS PREC $\delta$
$<$ PREC $\delta_{j-1}$

NO

CLA $-$ ZERO $\rightarrow M_{i1}$

FAM $- L_{a-1} \rightarrow M_{i1}$

OR

FSB $- L_{a-1} \rightarrow M_{i2}$

STD $- R_i \rightarrow M_{i3}$

$R_i \rightarrow L_{a-1}$

$R_{i+1} \rightarrow R_i$

$\beta - 1 \rightarrow \beta$

LOQ $- s_{\beta-1} \rightarrow M_{i1}$

CLA $- L_{a-1} \rightarrow M_{i2}$

TSX 1 EXP $\rightarrow M_{i3}$

STQ $- R_i \rightarrow M_{i4}$

$R_i \rightarrow L_{a-1}$

$R_{i+1} \rightarrow R_i$

$\beta - 2 \rightarrow \beta$

CLA $-$ ZERO $\rightarrow M_{i1}$

FAM $- s_{\beta-1} \rightarrow M_{i2}$

STO $- R_q \rightarrow M_{i3}$

$R_i \rightarrow s_{\beta-2}$

$\beta - 1 \rightarrow \beta$

YES

$\delta_j \rightarrow$ TEMP STORE

CLA $- L_{a-1} \rightarrow M_{i1}$

OP $- L_{a-3} \rightarrow M_{i2}$

STO $- R_i \rightarrow M_{i3}$

$R_i \rightarrow L_{a-3}$

$R_i + 1 \rightarrow R_i$

$a - 2 \rightarrow a$

$\delta_j \rightarrow AC$

START

START

OTHERS

Figure XII-2 - Compiler General Flow Diagram

B

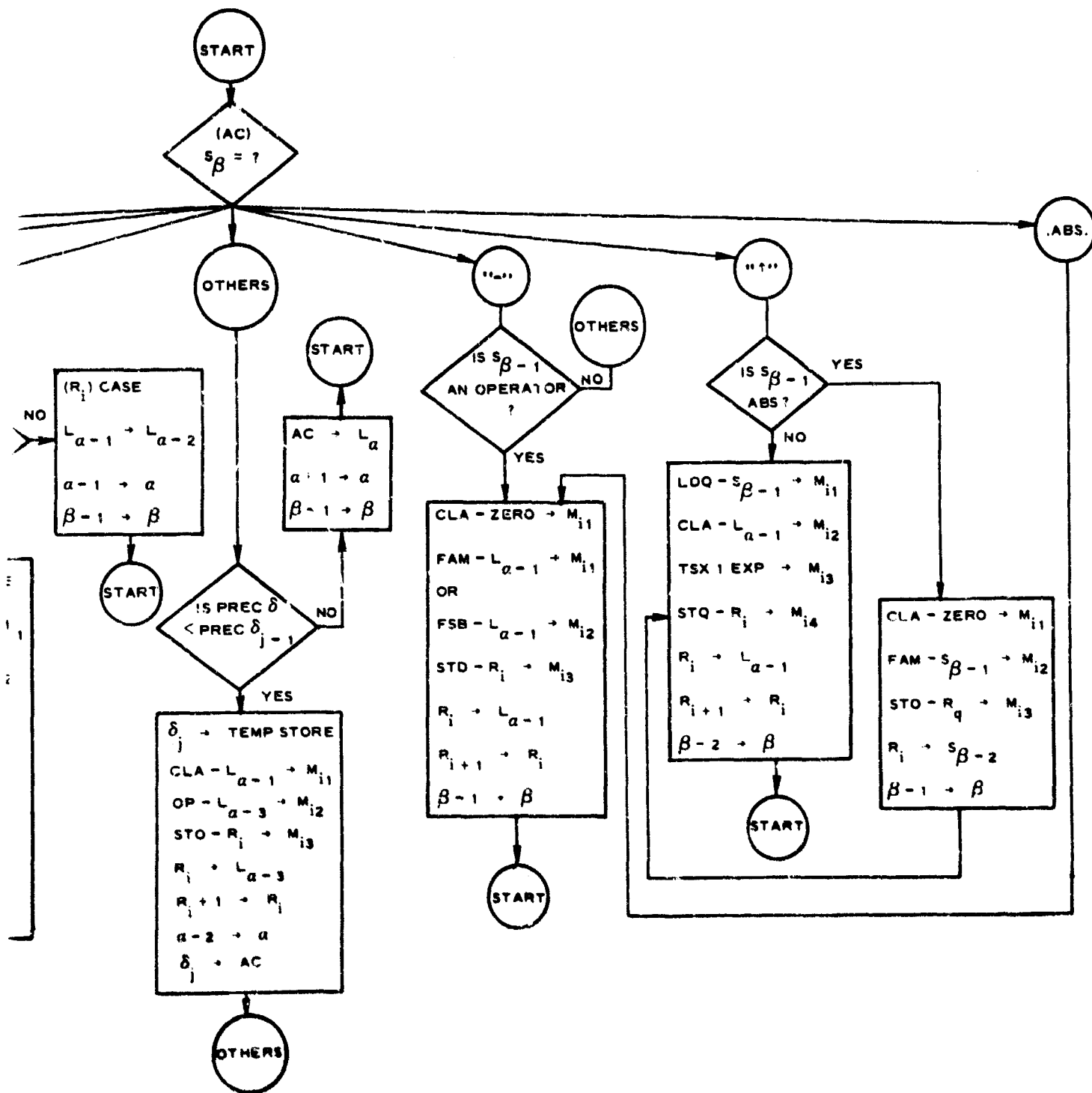examine the next item. If the absolute value is detected while examining an ↑-triple, then the tree mentioned instructions are formed before the ↑-triple instructions.

(2) Unary/Binary Minus

If a seven is detected in the decrement field of an item on the input string (SLIST) during the initial scan, the subroutine MINUS is entered. This subroutine will check the next item on the list $(S_{\beta-1})$ to see if it is an operator. If the next item $(S_{\beta-1})$ is not an operator, then the minus operation is binary and program control goes to the OTHERS subroutine.

If the next item $(S_{\beta-1})$ is an operator, then the minus is a unary operation and three instructions are formed: (1) CLA POOL, which makes the contents of the AC register equal to zero once this instruction is executed at object time; (2) FSB $L_{\alpha-1}$, which will subtract from zero (AC register) the contents of $L_{\alpha-1}$; and (3) STO $R_i$, which will store the result of the unary operation in an intermediate location.

Then $\alpha$ and $\beta$ are set to examine the next input item and compiler control returns to START.

(3) Exponentiation

If an eleven is detected in the decrement field of an item on the input string (SLIST) during the initial scan, the POWERS subroutine is entered. This subroutine checks the next operator $(S_{\beta-1})$ and if it is an absolute value operator performs a compilation of an absolute value operation first.

After the absolute value compilation or if $S_{\beta-1}$ is not an absolute value operator, four instructions are generated for EXP subroutine calling sequences:

1. LDQ $S_{\beta-1}$ - the value to be raised to a power is placed in the MQ register

2. CLA $L_{\alpha-1}$ - the exponent to raise a value to a power is placed in the AC register

3. TSX 1 EXP - transfer and set index 1 to the current object program address and go to subroutine EXP

4. STQ $R_i$ - store the result of EXP in an intermediate location.

Then $\alpha$ and 1 are set to examine the next item on the S list and compiler control returns to START.

(4) Parentheses

Once the operator "(" is detected (a four in the decrement field), compiler control stays in the LFTPRN loop until all operations within the parenthesis have been compiled. The loop stop code is of course ")" (a three in the decrement field), and at this time compiler control goes to RTPRN and sets $\alpha$ and $\beta$ to examine the next input item. The final resultant within the parenthesis $(R_p)$ is moved up one position on the L list by the RTPRN routine to completely eliminate the parenthesis. For an illustration of this case, see Page 316.

(5) Terminators

Once the - operator is detected (a two in the decrement field), compiler control stays in the TERM loop until all the operations of the expression have been compiled. The loop stop code for this routine is ⊣ (a one in the decrement field). Once the condition ⊢ $R_f$ ⊣ exists and there are no more expressions to be compiled, control goes to list and/or execute the program. For an illustration of this case, see Page 316.

(6) Others

The OTHERS subroutine is the most general subroutine in the

compiler and effectively the input string could be defined in such a manner that OTHERS would be the only subroutine necessary for the compiler. In the next section, the point is brought out as to how expressions could be written that require extensive use of the OTHERS routine. This subroutine first will compare the precedence of the current operator with the precedence of the previous operator; that is, $PREC(\delta_j) < PREC(\delta_{j-1})$. If the current operator precedence is less than the previous operator precedence, a set of object code instructions using the previous operator is formed. If precedence of the current operator is not less than the precedence of the previous operator, it is added to the intermediate (L) list and the next item on the S list is examined.

3. A SIMPLIFIED APPROACH TO COMPILING SUBSTITUTION EXPRESSIONS

    a. General

    In writing substitution statements, many compilers try to reduce the complexity of compilation by placing restrictions on the programmer. Sometimes, these restrictions are a set of programming rules that will discourage the use of parentheses or encourage the writing of unary operations in a prescribed manner. By using the expression on Page 288, the programmer could have conceivably written:

$$V = A*B + C*D \tag{1}$$

$$W = E \uparrow F + \phi . ABS. I \tag{2}$$

$$X = J K - L/M \tag{3}$$

$$Y = Q*R - P \tag{4}$$

$$Z = V*W + X/Y \tag{5}$$

Where $\phi$ is a location containing zero. It is noted that each simplified expression contains an odd number of items. As a general rule, when writing expressions in the above, simplified manner, the compilation

time is thought to be reduced considerably. Such is not the case when using this algorithm for compilation. The reason for this is due to the shorter loops (see Page 290) for

<div align="center">

Unary minus - 44 cycles ,

↑(POWERS) - 45 cycles ,

Normal absolute value - 37 cycles ,

</div>

whereas the general OTHERS loop requires 50 cycles. The OTHERS routine as discussed in greater detail on Page 294 is used for a general timing equation derivation later in this section.

b.  Compilation of Complex Expressions

A complete simulation of the compilation of the expression given on Page 288 is illustrated in great detail beginning on Page 316. The overall compilation time for the complex expression is found to be 1270 cycles or

$$T_{(total)} = 1270(2.18^a) \ \mu sec$$

$$= 2.3686 \ msec .$$

The reason for such detail is to give the reader an insight into what is involved in order to do a compilation.

c.  Derivation of a General Timing Equation for Compiling Simplified Expressions

From the simplified expressions on Page 295 and the flow diagram on Page 297, it can be noted that there are $(n - 1)/2$ operators, $(n - 1)/2$ operands, and $(n - 1)/2$ triples, where $n$ is the number of items in the expression. It is seen from the general timing equations beginning on Page 316 that the compiler requires 11 cycles to acknowledge and transfer an operand from the input string to the intermediate list so the time (in cycles) to transfer operands is

---

[a] The 7090 cycle time is 2.18 μsec per cycle.
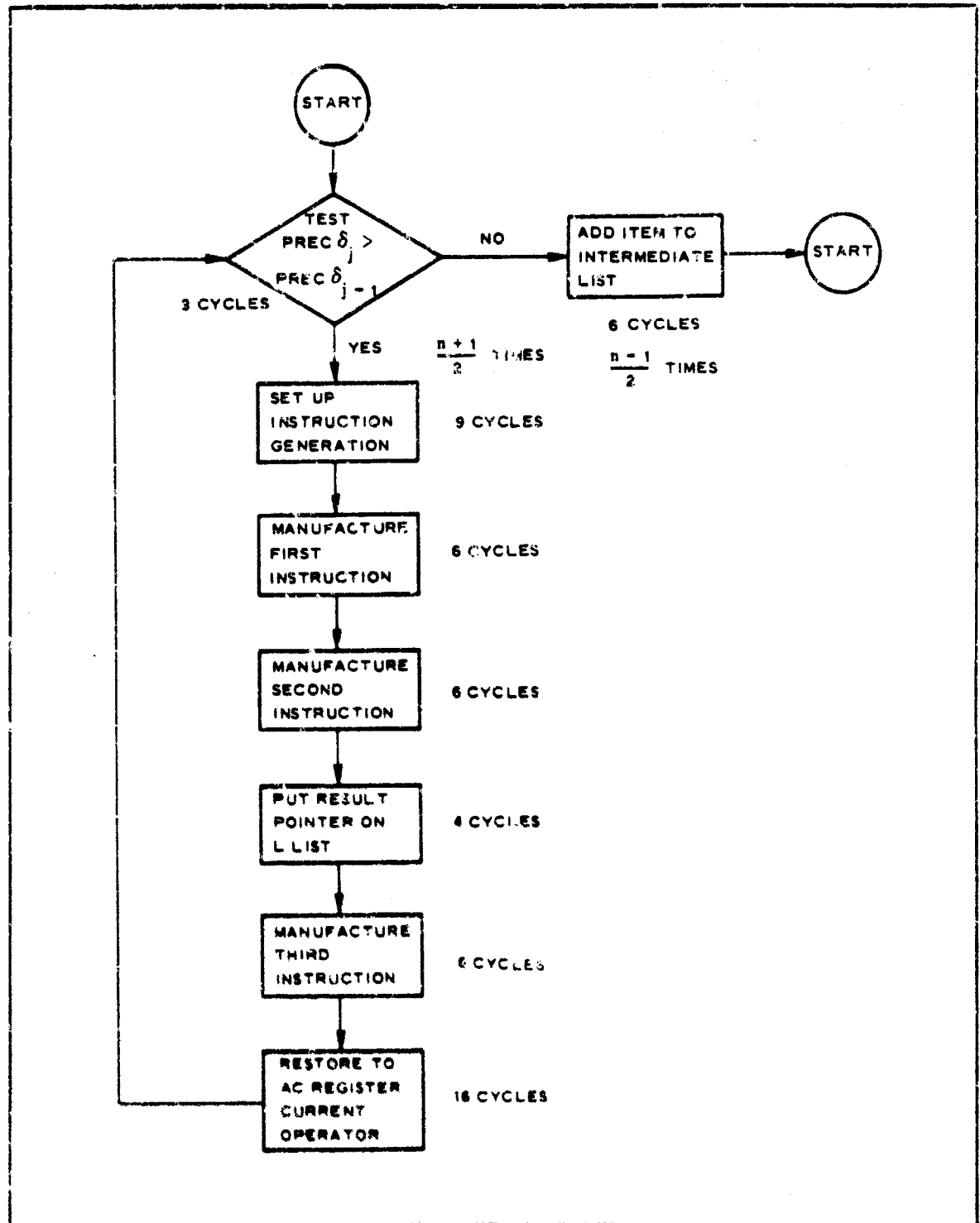
Figure XII-3 - An Arithmetic Opera. r General Flow Diagram

-297-

$$t_{(operands)} = 11\,\frac{n+1}{2}\;. \tag{6}$$

Using the data from the timing equations, it follows that there is generally

$$t_{(triple)} = 50\,\frac{n-1}{2} \tag{7}$$

cycles per simplified expression.

The timing equations show that 15 cycles per operator is required or

$$t_{(operators)} = \frac{15(n-1)}{2}\;. \tag{8}$$

Using the three equations, the total time for each expression is

$$t_{(total)} = t_{(operands)} + t_{(triples)} + t_{(operators)}$$

$$= \frac{11(N+1)}{2} + \frac{50(N-1)}{2} + \frac{15(N-1)}{2} = 38N - 27 \text{ cycles}\;. \tag{9}$$

If $N = 9$ as in the simplified expressions 1, 2, 3, or 5 on Page 295, then

$$t_{(total)}^{(1)} = 38(9) - 27 = 315 \text{ cycles}\;.$$

In simplified expression (4) on Page 293, $N = 7$ and

$$t_{(total)}^{(4)} = 38(7) - 27 = 239 \text{ cycles}$$

so the overall time required to compile the simplified expressions is:

$$T_{(total)} = t_{(total)}^{(1)} + t_{(total)}^{(2)} + t_{(total)}^{(3)} + t_{(total)}^{(4)} + t_{(total)}^{(5)}$$

$$= 315 + 315 + 315 + 239 + 315 = 1499 \text{ cycles}\;.$$

## 4. CHARTS, ASSEMBLY LISTING, AND TIMING EQUATIONS

The compiler flow charts, an assembly listing of a compiler, the general timing equations, and a simulation of a compilation are presented on the following pages.

Figure XII-4 - Compiler Flow Chart
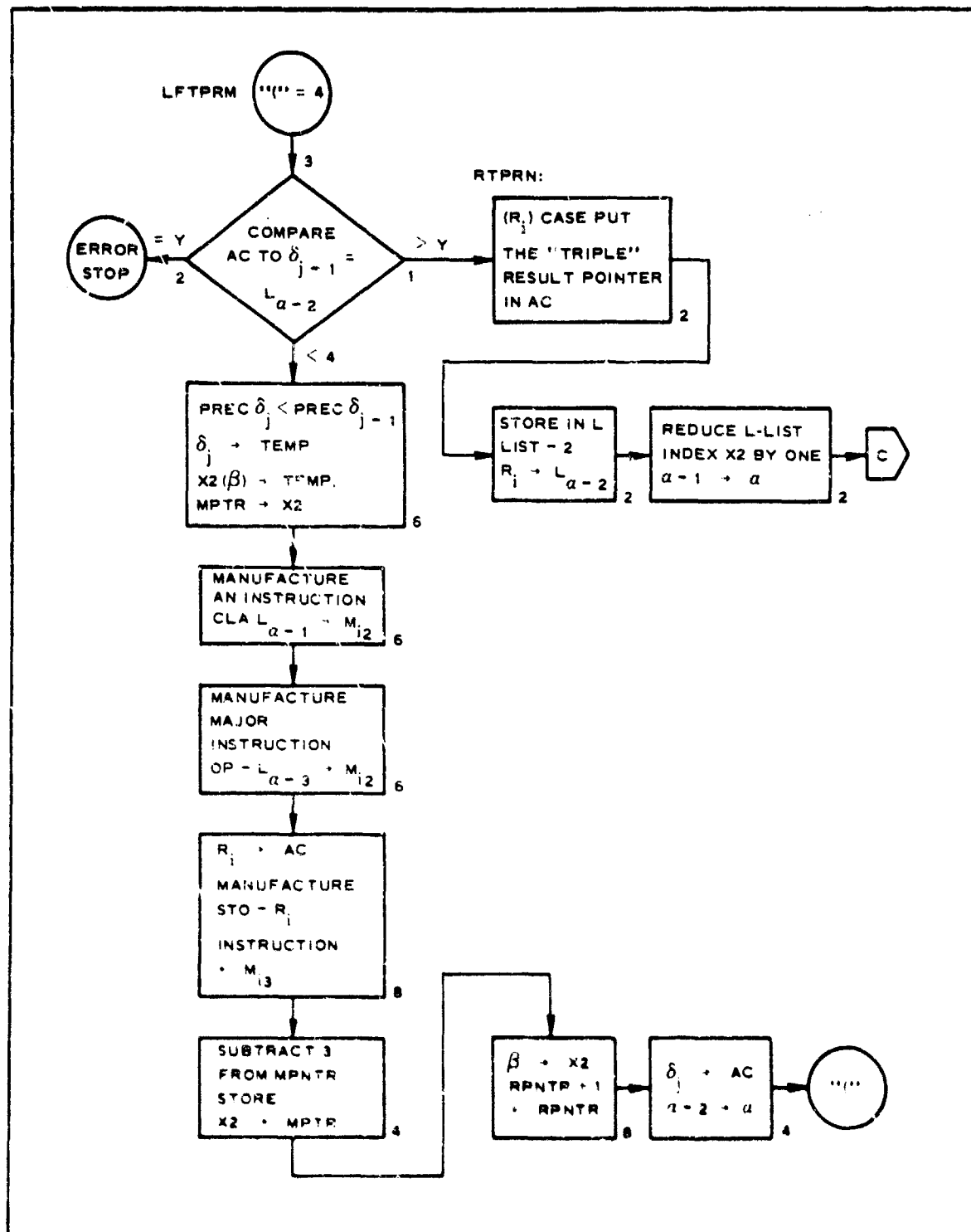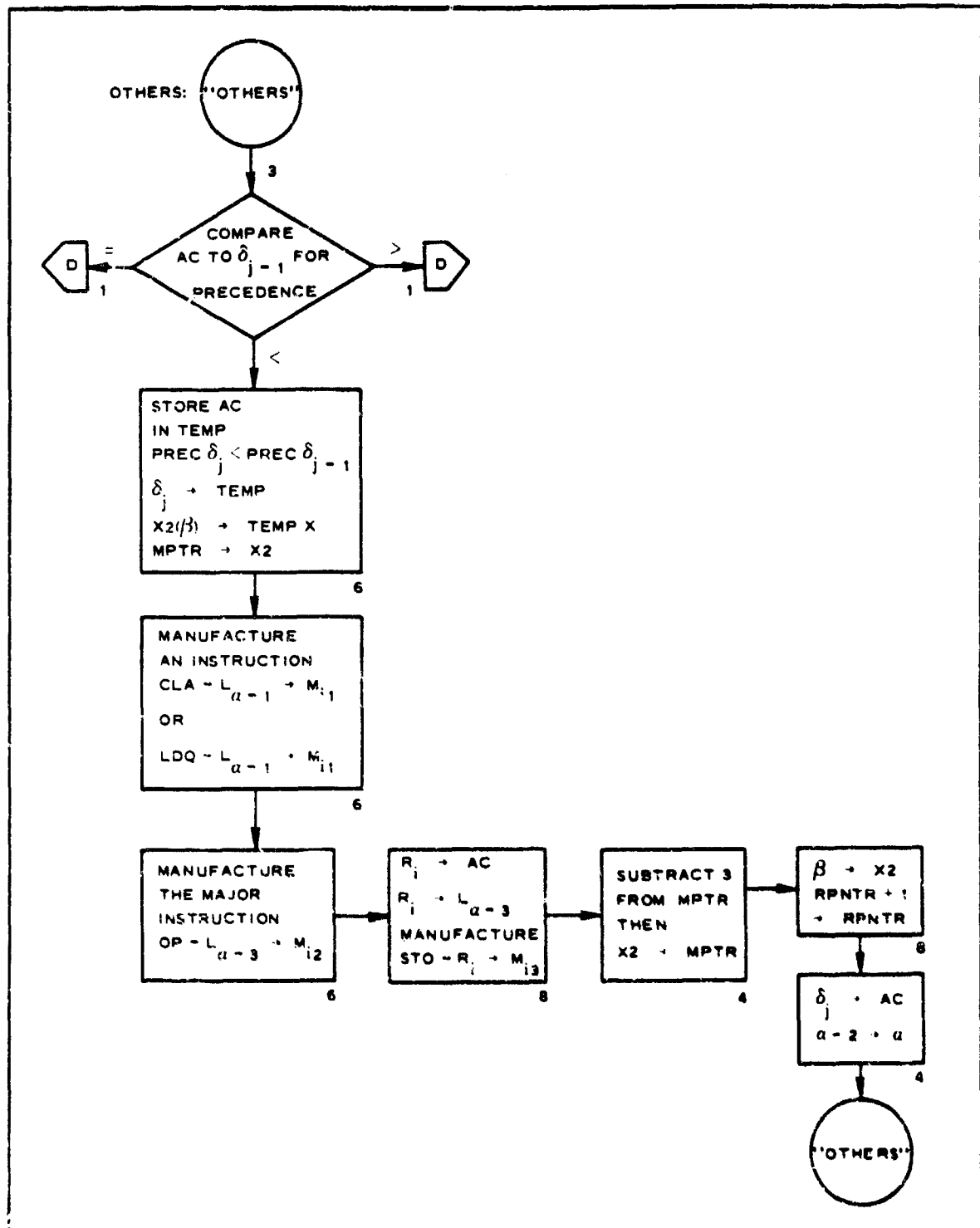
Figure XII-4 - Compiler Flow Chart (Continued)

Figure XII-4 - Compiler Flow Chart (Continued)
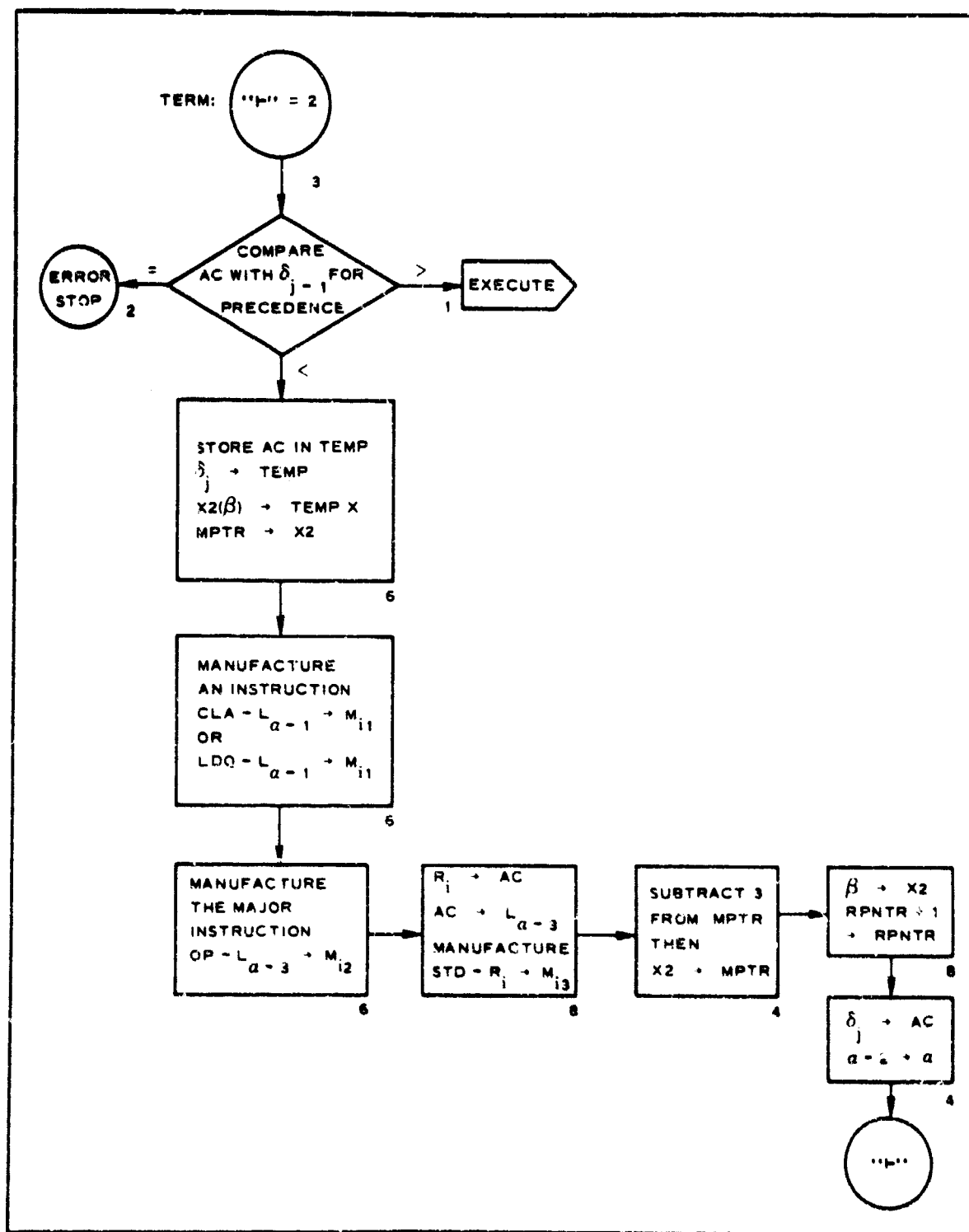
Figure XII-4 - Compiler Flow Chart (Continued)
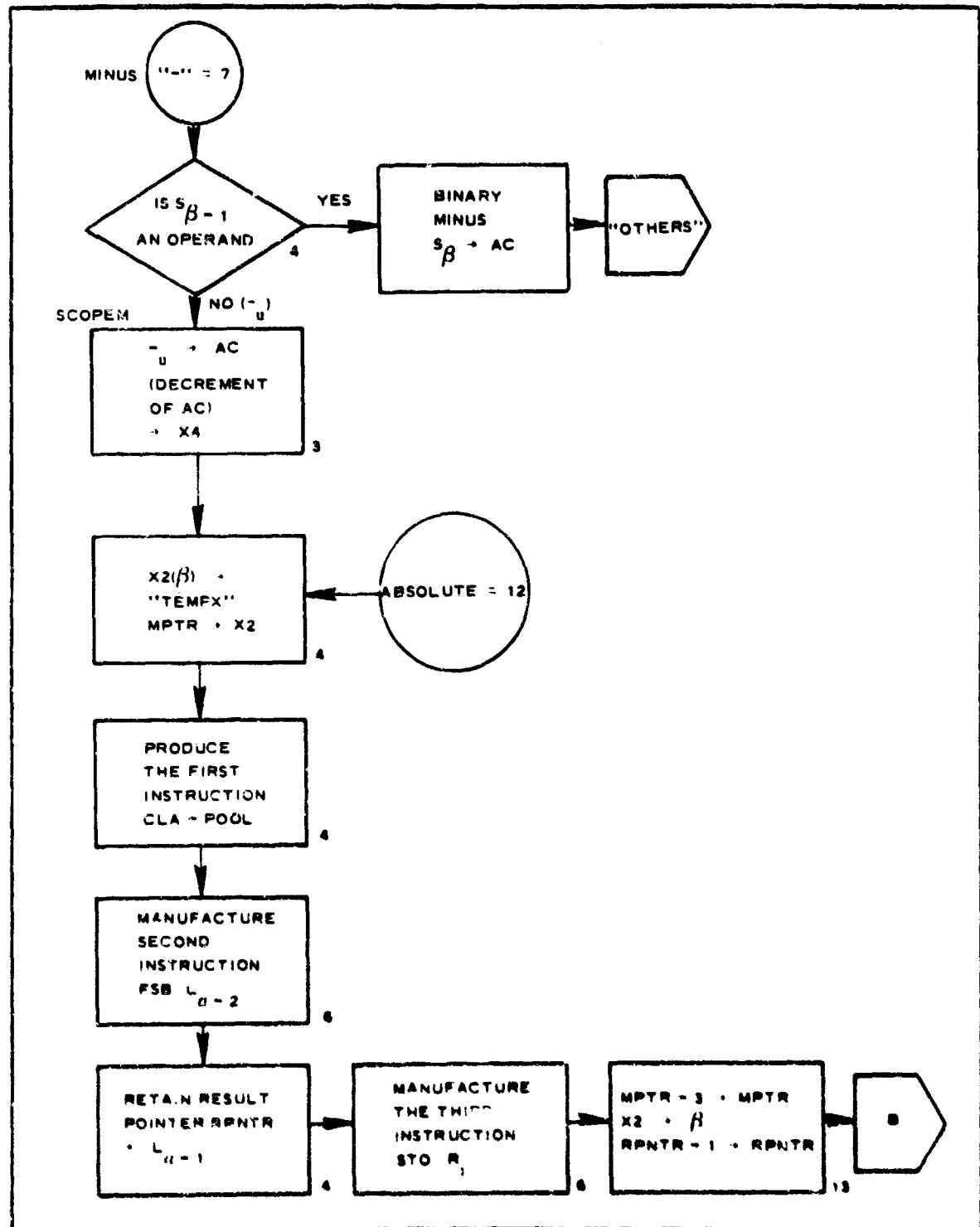
Figure XII-4 - Compiler Flow Chart (Continued)
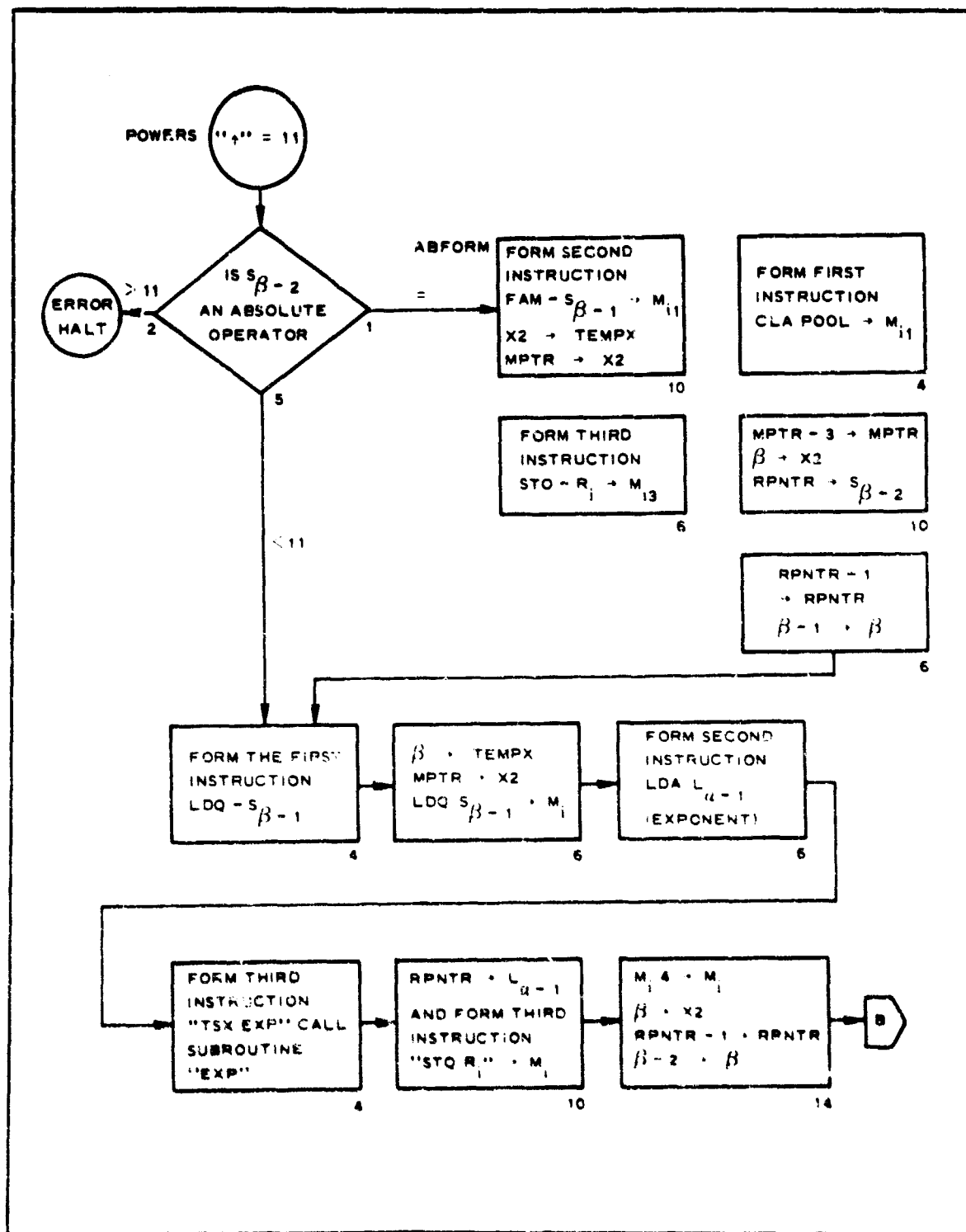
Figure XII-4 - Compiler Flow Chart (Continued)

## ASSEMBLY LISTING OF COMPILER (Continued)

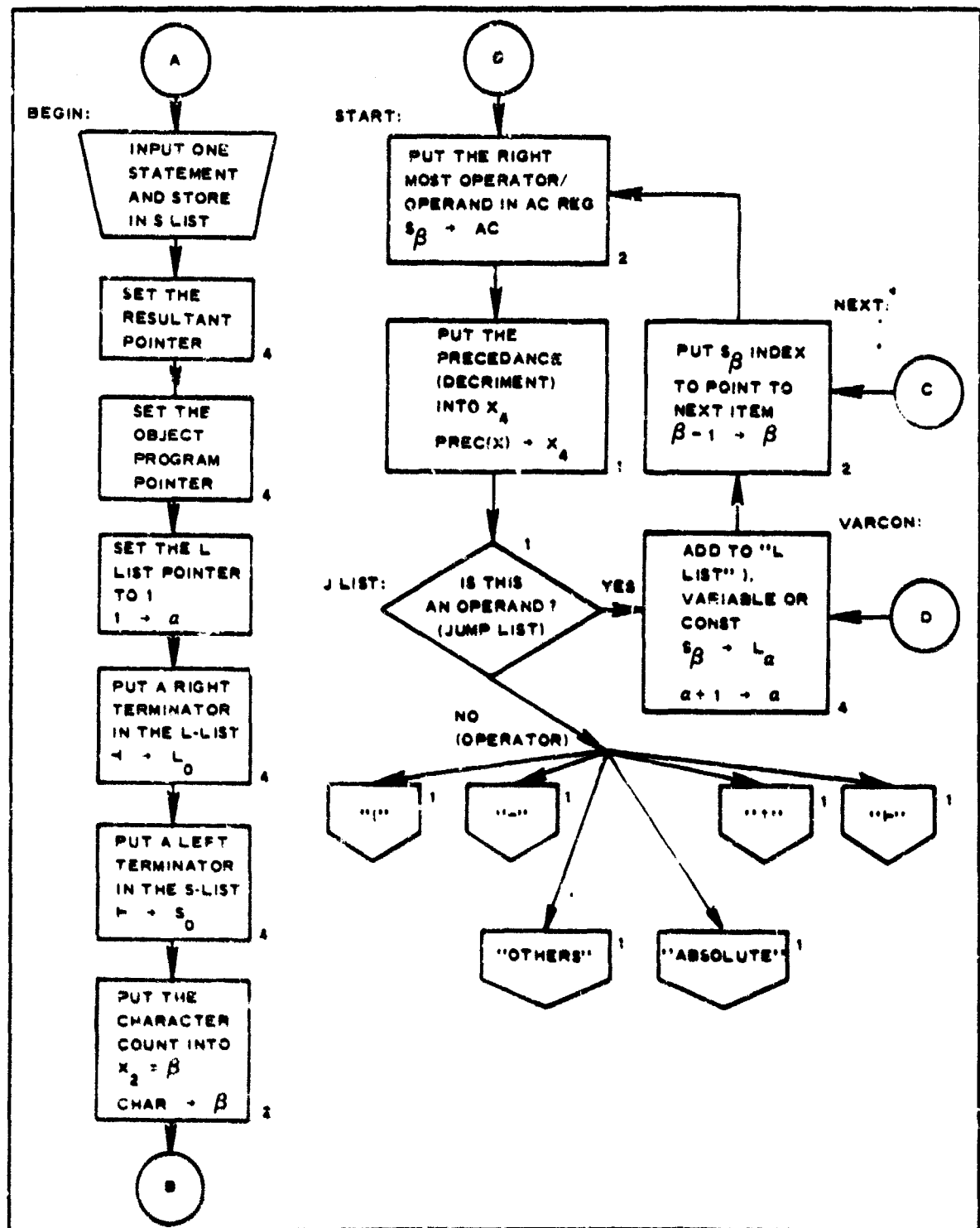| | OP | ADDRESS, TAG, DECREMENT | | | COMMENTS | |
|---|---|---|---|---|---|---|
| | | **8** | | | | |
| | TXI | *+1, 2, 3 | 2 | | $M + 3 \to M$ | 00490 |
| | SXA | MPTR, 2 | 2 | | $X2 \to MPTR$ | 00500 |
| | LXA | TEMPX, 2 | 2 | | $\beta \to X2$ | 00510 |
| | CLA | RPNTR | 2 | | | 00520 |
| | SUB | ONE | 2 | 16 | $-R_1 - 1 = -(R_1 + 1)$ | 00530 |
| | STO | RPNTR | 2 | | | 00540 |
| | CLA | TEMP | 2 | | | 00550 |
| | TXI | LFTPRN, 1, 2 | 2 | | $\alpha + 2 \to \alpha$ | 00560 |
| RTPRN | CLA | LLIST -1, 1 | 2 | | $(R_2)$ CASE | 00570 |
| | STO | LLIST -2, 1 | 2 } 6 | | | 00580 |
| | TXI | NEXT, 1, 1 | 2 | | $\alpha - 1 \to \alpha$ | 00590 |
| OTHERS | CAS | LLIST -2, 1 | 3 | | | 00600 |
| | TRA | VARCON | 1 | | AC > Y GO ADD | 00610 |
| | TRA | VARCON | 1 | | AC = Y TO LIST | 00620 |
| | STO | TEMP | 2 | | $\delta_j \to TEMP$ | 00630 |
| | CLA | LLIST -2, 1 | 2 | | | 00640 |
| | PDC | $\phi$, 4 | 1 } 9 | | AC (DECREMENT COMPLEMENT) X4 | 00660 |
| | SXA | TEMPX, 2 | 2 | | $X2(\beta) \to TEMPX$ | 00670 |
| | LXA | MPTR, 2 | 2 | | $MPTR \to X2$ | 00680 |
| | CLA | DUMMYA, 4 | 2 | | | 00690 |
| | ADM | LLIST -1, 1 | 2 } 6 | | MANUFACTURE AN | 00700 |
| | STO | $\phi$, 2 | 2 | | INSTRUCTION | 00710 |
| | CLA | DUMMYB, 4 | 2 | | MAIN | 00720 |
| | ADM | LLIST -3, 1 | 2 } 6 | | INSTRUCTION | 00730 |
| | STO | 1, 2 | 2 | | | 00740 |
| | CLA | RPNTR | 2 | | $R_i \to L_{\alpha-3}$ | 00750 |
| | STO | LLIST -3 | 2 } 4 | | | 00760 |
| | SSP | 3 | 2 | | MAKE AC+ | 00770 |
| | ADD | DUMMYR, 4 | 2 } 4 | | $STO - R_i \to M_{13}$ | 00780 |
| | STO | 2, 2 | 2 | | | 00790 |
| | TXI | *+1, 2, -3 | 2 | | $X2+3 \to X2$ | 00800 |
| | SXA | MPTR, 2 | 2 | | | 00810 |
| | LXA | TEMPX, 2 | 2 | | | 00820 |
| | CLA | RPNTR | 2 | | | 00830 |
| | SUB | ONE | 2 } 16 | | $-R_2 - 1 = -(R_2 + 1)$ | 00840 |
| | STO | RPNTR | 2 | | | 00850 |
| | CLA | TEMP | 2 | | $\delta_j \to AC$ | 00860 |
| | TXI | OTHERS, 1, 2 | 3 | | $\alpha - 2 \to \alpha$ | 00870 |
| TERM | CAS | LLIST -2, 1 | 3 | | TEST PREC | 00880 |
| | TRA | EXECUT | 1 | | LIST AND/OR EXECUTE $\to R_\gamma$ | 00890 |
| | HTR | BEGIN | 3 | | ERROR 2 > 5 | 00900 |
| | STO | TEMP | 2 | | $\delta_j \to TEMP$ | 00910 |
| | CLA | LLIST -2, 1 | 2 | | | 00920 |
| | PDC | $\phi$, 4 | 1 | | | 00930 |
| | SXA | TEMPX, 2 | 2 } 15 | | $X2(\beta) \to TEMPX$ | 00940 |
| | LXA | MPTR, 2 | 2 | | $MPTR \to X2$ | 00950 |
| | CLA | DUMMYA, 4 | 2 | | | 00960 |
| | ADM | LLIST -3, 1 | 2 | | MANUFACTURE | 00970 |
| | STO | $\phi$, 2 | 2 | | | 00980 |

-306-

## ASSEMBLY LISTING OF COMPILER (Continued)

| | OP | ADDRESS, TAG, DECREMENT | | | COMMENTS | |
|---|---|---|---|---|---|---|
| | | | **8** | | | |
| | CLA | DUMMYB, 4 | 2 | | | 00990 |
| | ADM | LLIST -1, 1 | 2 | 6 | STORE FINAL RESULT IN | 01000 |
| | STO | 1, 2 | 2 | | SUBSCRIPTED VARIABLE | 01010 |
| | CLA | RPNTR | 2 | 4 | | 01020 |
| | STO | LLIST -3 | 2 | | $R_2 \to L_{\alpha+3}$ | 01030 |
| | SSP | 3 | 2 | | MAKE AC+ | 01040 |
| | ADD | DUMMYR, 4 | 2 | 6 | SAVE NEXT $R_1$ LOCATION | 01050 |
| | STO | 2, 2 | 2 | | FOR THE NEXT EXPRESSION | 01060 |
| | TXI | *+1, 2, -3 | 2 | | $X2 - 3 \to X2$ | 01070 |
| | SXA | MPTR, 2 | 2 | | $X2 \to MPTR$ | 01080 |
| | LXA | TEMPX, 2 | 2 | | $\beta \to X2$ | 01090 |
| | CLA | RPNTR | 2 | 16 | | 01100 |
| | SUB | ONE | 2 | | INCREASE $R_2$ POINTER | 01110 |
| | STO | RPNTR | 2 | | | 01120 |
| | CLA | TEMF | 2 | | $\delta_j \to AC$ | 01130 |
| | TXI | TERM, 1, 2 | 2 | | $\alpha - 2 \to \alpha$ | 01140 |
| MINUS | CLA | SLIST -1, 2 | 2 | | $\delta_{\beta-1} \to AC$ | 0115C |
| | TPL | SCOPEM | 2 | | IF (-) OPERAND IF (+) OPERATOR | 01160 |
| | CLA | SLIST, 2 | 2 | 5 | $\delta_\beta \to AC$ | 01170 |
| | TRA | OTHERS | 1 | | | 01180 |
| SCOPEM | CLA | UNMIN | 2 | | $PREC \cdot_u \to AC$ | 01190 |
| | PDC | 4, 4 | 1 | 7 | $DEC \cdot_u \to X4$ | 01200 |
| | SXA | TEMPX, 2 | 2 | | $X2(\beta) \to TEMPX$ | 01210 |
| | LXA | MPTR, 2 | 2 | | $MPTR \to X2$ | 01220 |
| | CLA | CLEAR | 2 | 4 | $CLA \cdot POOL \to M_{1L}$ | 01230 |
| | STO | 4, 2 | 2 | | | 01240 |
| | CLA | DUMMYB, 4 | 2 | | | 01250 |
| | ADM | LLIST -1, 1 | 2 | 6 | $FSB\ L_{\alpha-1} \to M_{12}$ | 01260 |
| | STO | 1, 2 | 2 | | | 01270 |
| | CLA | RPNTR | 2 | 4 | $R_1 \to AC$ | 01280 |
| | STO | LLIST -1, 1 | 2 | | $R_1 \to L_{\alpha-1}$ | 01290 |
| | SSP | 3 | 2 | | | 01300 |
| | ADD | DUMMYR, 4 | 2 | 6 | $STO \cdot R_1 \to M_{13}$ | 01310 |
| | STO | 2, 2 | 2 | | | 01320 |
| | TXI | *+1, 2, -3 | 2 | | | 01330 |
| | SXA | MPTR, 2 | 2 | | $X2 \to MPTR$ | 01340 |
| | LXA | TEMPX, 2 | 2 | | | 01350 |
| | CLA | RPNTR | 2 | 13 | | 01360 |
| | SUB | ONE | 2 | | INCREASE $R_1$ POINTER | 01370 |
| | STO | RPNTR | 2 | | | 01380 |
| | TXI | START, 2, 1 | 1 | | $\beta - 1 \to \beta$ | 01390 |
| POWERS | CLA | SLIST -2, 2 | 2 | | $\delta_{j+1} \to AC$ | 01400 |
| | CAS | ABSVAL | 3 | | TEST .ABS. | 01410 |
| | HTR | BEGIN | 2 | | ERROR HALT | 01420 |
| | TRA | ABFORM | 1 | | | 01430 |
| | CLA | DUMMYA, 4 | 2 | 4 | | 01440 |
| | ADM | SLIST -1, 2 | 2 | | | 01450 |
| | SXA | TEMPX, 2 | 2 | | $X2(\beta) \to TEMPX$ | 01460 |
| | LXA | MPTR, 2 | 2 | 6 | $MPTR \to X2$ | 01470 |
| | STO | 4, 2 | 2 | | $LDQ \cdot \delta_{\beta-1} \to M_{11}$ | 01480 |

## ASSEMBLY LISTING OF COMPILER (Continued)

8

| | OP | ADDRESS, TAG, DECREMENT | | COMMENTS | |
|---|---|---|---|---|---|
| | CLA | DUMMYB, 4 | 2 | | 01490 |
| | ADM | LLIST -1, 1 | 2 }6 | $CLA - L_{\alpha-1} \rightarrow M_{12}$ | 01500 |
| | STO | 1, 2 | 2 | | 01510 |
| | CLA | TRANSX | 2 | $TSX - EXP \rightarrow M_{13}$ | 01520 |
| | STO | 2, 2 | 2 }4 | | 01530 |
| | CLA | RPNTR | 2 | | 01540 |
| | STO | LLIST -1, 1 | 2 }4 | $R_i \rightarrow L_{\alpha-1}$ | 01550 |
| | SSP | 3 | 2 | MAKE+ | 01560 |
| | ADD | DUMMYR, 4 | 2 }6 | $STQ - R_i \rightarrow M_{14}$ | 01570 |
| | STO | 3, 2 | 2 | | 01580 |
| | TXI | *+1, 2, 4 | 2 | | 01590 |
| | SXA | MPTR, 2 | 2 | $X2 \rightarrow MPNTR$ | 01600 |
| | LXA | TEMPX, 2 | 2 | $\beta \rightarrow X2$ | 01610 |
| | CLA | RPNTR | 2 }4 | | 01620 |
| | SUB | ONE | 2 | INCREASE $R_i$ POINTER | 01630 |
| | STO | RPNTR | 2 | | 01640 |
| | TXI | STAPT, 2, 2 | 2 | $\beta - 2 \rightarrow \beta$ | 01650 |
| ABFORM | CLA | DUMMYB, +12 | 2 | | 01660 |
| | ADM | SLIST -1, 2 | 2 | $FAM - S_{\beta-1} \rightarrow M_{12}$ | 01670 |
| | SXA | TEMPX, 2 | 2 }10 | | 01680 |
| | LXA | MPTR, 2 | 2 | | 01690 |
| | STO | 1, 2 | 2 | | 01700 |
| | CLA | CLEAR | 2 | $CLA - POOL \rightarrow M_{11}$ | 01710 |
| | STO | 0, 2 | 2 }4 | | 01720 |
| | C | DUMMYR, +12 | 2 | | 01730 |
| | | RPNTR | 2 }6 | $STO - R_i \rightarrow M_{13}$ | 01740 |
| | STO | 2, 2 | 2 | | 01750 |
| | TXI | *+1, 2, -3 | 2 | | 01760 |
| | SXA | MPTR, 2 | 2 | | 01770 |
| | LXA | TEMPX, 2 | 2 }10 | | 01780 |
| | CLA | RPNTR | 2 | | 01790 |
| | STO | SLIST -2, 2 | 2 | $R_i \rightarrow S_{\beta-1}$ | 01800 |
| | SUB | ONE | 2 | | 01810 |
| | STO | RPNTR | 2 | | 01820 |
| | TXI | POWERS +4, 2, 1 | 2 | | 01830 |
| EXECUT | TSX | DATAIN, 1 | | $EXECUT \rightarrow X1$ | 01840 |
| | TRA | MLIST | | | 01850 |
| MLIST | BSS | 1 | | OBJECT PROGRAM | 01860 |
| RST | OCT | -100 | | RELATIVE DISPLACEMENT | 01870 |
| RPNTR | BSS | | | RESULT POINTER | 01880 |
| IST | | MLIST | | SET FOR MPNTR | 01890 |
| MPTR | BSS | | | OBJECT PROGRAM POINTER | 01900 |
| RTERM | OCT | ► OP, 0 1 | | RIGHT TERMINATOR | 01910 |
| DUMMYA | NOP | POOL | | VARIABLE OR CONSTANT ◂ | 01911 |
| | NOP | POOL | | ◂ | 01912 |
| | NOP | POOL | | ► | 01913 |
| | NOP | POOL | | ) | 01914 |
| | NOP | POOL | | ( | 01915 |
| | CLA | POOL | | ٠ | 01916 |

-308-

## ASSEMBLY LISTING OF COMPILER (Continued)

|  | 8 |  |  |  |
|---|---|---|---|---|
|  | OP | ADDRESS, TAG, DECREMENT | COMMENTS |  |
|  | CLA | POOL | $+$ POOL $+ L_{\alpha-1}$ | 01917 |
|  | CLA | POOL | $-$ | 01918 |
|  | CLA | POOL | / | 01919 |
|  | LDQ | POOL | $*$ | 01920 |
|  | CLA | POOL | $-_u$ | 01930 |
|  | LDQ | POOL | $\uparrow$ | 01940 |
|  | CLA | POOL | .ABS. | 01950 |
| DUMMYB | NOP | POOL | VARIABLE/CONSTANT | 01960 |
|  | NOP | POOL | $\dashv$ | 01970 |
|  | NOP | POOL | $\vdash$ | 01980 |
|  | NOP | POOL | ) | 01990 |
|  | NOP | POOL | ( | 02000 |
|  | STO | POOL | $=$ | 02010 |
|  | FAD | POOL | $+$ POOL $+ L_{\alpha-3}$ | 02020 |
|  | FSB | POOL | $-$ | 02030 |
|  | FPP | POOL | / | 02040 |
|  | FMP | POOL | $*$ | 02050 |
|  | FSB | POOL | $-_u$ | 02060 |
|  | LDA | POOL | $\uparrow$ | 02070 |
|  | FAM | POOL | .ABS. | 02080 |
| DUMMYR | NOP | POOL | VARIABLE/CONSTANT | 02090 |
|  | NOP | POOL | $\dashv$ | 02100 |
|  | NOP | POOL | $\vdash$ | 02110 |
|  | NOP | POOL | ) | 02120 |
|  | NOP | POOL | ( | 02130 |
|  | NOP | POOL | $+$ POOL $+ 100(R_i)$ | 02140 |
|  | STO | POOL | $+$ | 02150 |
|  | STO | POOL |  | 02160 |
|  | STQ | POOL | / | 02170 |
|  | STQ | POOL | $*$ | 02180 |
|  | STQ | POOL | $-_u$ | 02190 |
|  | STQ | POOL | $\uparrow$ | 02200 |
|  | STO | POOL | ABS. | 02210 |
| LIST | BSS | 100 | INTERMEDIATE STORAGE | 02220 |
| LTERM |  | $\vdash$ OP, 0, 2 | LEFT TERMINATOR OPERATOR | 02230 |
| SLIST | BSS | 100 | INPUT STRING | 02240 |
| CHAR | BSS |  | CHARACTER COUNT | 02250 |
| TEMP | BSS |  | TEMPORARY STORAGE | 02260 |
| TEMPX | BSS |  | INDEX STORAGE | 02270 |
| ONE | OCT | 1 |  | 02280 |
| UNMIN |  | $-_u$ OP 0 10 | UNARY OPERATOR | 02290 |
| CLEAR | CLA | POOL |  | 02300 |
| POOL | ZRO |  | ZERO | 02310 |
|  | BSS | 200 | CONSTANT/VARIABLE/RESULTS | 02320 |
| ABSVAL |  | ABS OP 0 12 |  | 02340 |
| TRANSX | TSX | EXP | TRANSFER TO SUBROUTINE | 02350 |
| INPUT | REM | INPUT ONE STATEMENT | ROUTINE | 02360 |
| DATAIN | REM | READ DATA ROUTINE |  | 02370 |
|  | TRA | 1, 1 | GO TO L+1 (MAIN PROGRAM) | 02380 |

## GENERAL TIMING EQUATIONS

### Initialization

Time to set the triple resultant pointer and object program pointer:

$$BEGIN_{00010} - BEGIN_{00040} = 8 \text{ cycles}$$

Time to set the program to compile one expression:

$$SET_{00050} - SET_{00110} = 12 \text{ cycles}$$

Time to acknowledge one item:

$$START_{00120} - JLIST_{00150-00260} = 5 \text{ cycles}$$

### Variable or Constant

Time to transfer a variable or constant from the SLIST to the LLIST:

$$VARCON_{00270} - NEXT_{00290} = 6 \text{ cycles}$$

Time to discover $PREC("(") < PREC(\delta_{j-1})$

### Left Parenthesis

$$LFTPRN_{00300} = 3 \text{ cycles}$$

Time to manufacture first instruction:

$$LFTPRN_{00330} - LFTPRN_{00400} = 15 \text{ cycles}$$

Time to manufacture second instruction:

$$LFTPRN_{00410} - LFTPRN_{00430} = 6 \text{ cycles}$$

Time to STORE resultant pointer in LLIST -3

$$\text{LFTPRN}_{00440} - \text{LFTPRN}_{00450} = 4 \text{ cycles}$$

Time to manufacture third instruction:

$$\text{LFTPRN}_{00460} - \text{LFTPRN}_{00480} = 6 \text{ cycles}$$

Time to increase $R_i$ pointer by one, object program counter by three, and decrement $\alpha$ by two:

$$\text{LFTPRN}_{00490} - \text{LFTPRN}_{00560} = 16 \text{ cycles}$$

Time to form a triple set of instructions:

$$\Sigma\text{LFTPRN}_{00300} - \text{LFTPRN}_{00560} = 50 \text{ cycles}$$

Time to put $R_i$ at $L_{\alpha-1}$ back to $L_{\alpha-2}$ for $(R_i)$ case

$$\text{RTPRN}_{00570} - \text{RTPRN}_{00590} = 11 \text{ cycles}$$

## Others

Time to discover $\text{PREC}(\delta_j) < \text{PREC}(\delta_{j-1})$:

$$\text{OTHERS}_{00600} = 3 \text{ cycles} \qquad \text{if yes}$$

$$4 \text{ cycles} \qquad \text{if no}$$

Time to manufacture the first instruction:

$$\text{OTHERS}_{00630} - \text{OTHERS}_{00710} = 15 \text{ cycles}$$

Time to manufacture the second instruction:

$$\text{OTHERS}_{00720} - \text{OTHERS}_{00740} = 6 \text{ cycles}$$

Time to store resultant pointer in LLIST -3:

$$\text{OTHERS}_{00750} - \text{OTHERS}_{00760} = 4 \text{ cycles}$$

Time to manufacture the third instruction:

$$OTHERS_{00770} - OTHERS_{790} = 6 \text{ cycles}$$

Time to increase $R_i$ pointer by one, object program pointer by three, and decrement $\alpha$ by two

$$OTHERS_{00800} - OTHERS_{00870} = 16 \text{ cycles}$$

Time to form a triple set of instructions

$$\Sigma OTHERS_{00600} - OTHERS_{00870} = 50 \text{ cycles}$$

## Terminator

Time to test $PREC(\cdot) < PREC(\delta_{j-1})$:

$$TERM_{00880} = 3 \text{ cycles}$$

Time to manufacture the first instruction:

$$TERM_{00910} - TERM_{00980} = 15 \text{ cycles}$$

Time to manufacture the second instruction:

$$TERM_{00990} - TERM_{01010} = 6 \text{ cycles}$$

Time to put the $R_1$ pointer into LLIST -3:

$$TERM_{01020} - TERM_{01030} = 4 \text{ cycles}$$

Time to manufacture the third instruction:

$$TERM_{01040} - TERM_{01060} = 6 \text{ cycles}$$

Time to increase $R_i$ pointer by one, object program pointer by three, and decrement $\alpha$ by two:

$$TERM_{01070} - TERM_{01140} = 16 \text{ cycles}$$

Time to form a triple set of instructions:

$$\Sigma TERM_{00880} - TERM_{01140} = 50 \text{ cycles}$$

## Minus

Time to discover binary minus

$$MINUS_{01150} - MINUS_{01180} = 7 \text{ cycles}$$

Time to form binary minus triple set of instructions:

$$MINUS_{01150} - OTHERS_{00870} = 50 + 7 = 57 \text{ cycles}$$

Time to discover unary minus

$$MINUS_{01150} - MINUS_{01160} = 4 \text{ cycles}$$

Time to manufacture the first instruction:

$$SCOPEM_{01190} - SCOPEM_{01240} = 11 \text{ cycles}$$

Time to manufacture the second instruction:

$$SCOPEM_{01250} - SCOPEM_{01270} = 6 \text{ cycles}$$

Time to put the resultant point into LLIST -1:

$$SCOPEM_{01280} - SCOPEM_{01290} = 4 \text{ cycles}$$

Time to manufacture the third instruction:

$$SCOPEM_{01300} - SCOPEM_{01320} = 6 \text{ cycles}$$

Time to increase $R_i$ pointer by one, object program counter by three, and decriment $\beta$ by one:

$$SCOPEM_{01330} - SCOPEM_{01390} = 13 \text{ cycles}$$

Time to form a unary minus triple set of instructions:

$$\Sigma MINUS_{01150} - SCOPEM_{01390} = 44 \text{ cycles}$$

## Powers

Time to test $S_{\beta-2}$ for absolute value:

$$POWERS_{01400} - POWERS_{01410} = 5 \text{ cycles}$$

Time to manufacture the first instruction:

$$POWERS_{1440} - POWERS_{01480} = 10 \text{ cycles}$$

Time to manufacture the second instruction:

$$POWERS_{01490} - POWERS_{01510} = 6 \text{ cycles}$$

Time to manufacture the third instruction:

$$POWERS_{1520} - POWERS_{1530} = 4 \text{ cycles}$$

Time to put the $R_i$ pointer on LLIST -1:

$$POWERS_{01560} - POWERS_{0158} = 6 \text{ cycles}$$

Time to increase the $R_i$ pointer by one, the program pointer by four, and decrease $\beta$ by two:

$$POWERS_{01590} - POWERS_{01650} = 14 \text{ cycles}$$

Time to form a calling sequence of four instructions:

$$\Sigma POWERS_{01400} - POWERS_{01550} = 45 \text{ cycles}$$

## .ABS. V ↑ V Condition

Time if $S_{\beta-2} = $ .ABS.:

$$POWERS_{01400} - POWERS_{01430} = 6 \text{ cycles}$$

Time to form the first instruction:

$$\text{ABFORM}_{01650} - \text{ABFORM}_{01700} = 10 \text{ cycles}$$

Time to form the second instruction:

$$\text{ABFORM}_{01710} - \text{ABFORM}_{01720} = 4 \text{ cycles}$$

Time to form the third instruction:

$$\text{ABFORM}_{01730} - \text{ABFORM}_{01750} = 6 \text{ cycles}$$

Time to put $R_i$ on SLIST -2 and increase the object program counter by three:

$$\text{ABFORM}_{01760} - \text{ABFORM}_{01800} = 10 \text{ cycles}$$

Time to increase $R_i$ pointer by one and decrease $\beta$ by one:

$$\text{ABFORM}_{01810} - \text{ABFORM}_{01830} = 6 \text{ cycles}$$

Time to form a triple set of instructions for this case of absolute value:

$$\Sigma\text{POWERS}_{01400} - \text{ABFORM}_{01830} = 42 \text{ cycles}$$

Normal Absolute Value

$$\Sigma\text{SCOPEM}_{01210} - \text{SCOPEM}_{01390} = 37 \text{ cycles}$$

## SIMULATION OF A COMPILATION

### Status of SList

| $S_\beta$ | Accumulative time to reference item (cycles) | $S_\beta$ | Accumulative time to reference item (cycles) |
|---|---|---|---|
| ⊢ | 1216 | ) | 597 |
| Z | 1205 | + | 532 |
| = | 1090 | ( | 465 |
| ( | 973 | J | 420 |
| A | 962 | ↑ | 411 |
| * | 947 | K | 400 |
| B | 936 | - | 328 |
| + | 871 | L | 317 |
| C | 860 | / | 302 |
| * | 845 | M | 291 |
| D | 834 | ) | 280 |
| ) | 823 | / | 265 |
| * | 808 | ( | 198 |
| ( | 741 | - | 149 |
| E | 696 | P | 138 |
| ↑ | 687 | + | 73 |
| F | 676 | Q | 62 |
| + | 661 | ◊ | 47 |
| . ABS. | 619 | R | 36 |
| I | 608 | ) | 25 |

### Status of LList at Various Intervals

| $\alpha$ | $L_\alpha$ | Accumulative time (cycles) $S_\beta \rightarrow L_\alpha$ | $\alpha$ | $L_\alpha$ | $S_\beta \rightarrow L_\alpha$ | $\alpha$ | $L_\alpha$ | $S_\beta \rightarrow L_\alpha$ |
|---|---|---|---|---|---|---|---|---|
| 0 | ⊣ | 20 | 0 | ⊣ | 20 | 0 | ⊣ | 20 |
| 1 | ) | 31 | 1 | ) | 31 | 1 | ) | 31 |
| 2 | R | 42 | 2 | $R_1$ | 101 | 2 | $R_1$ | 101 |
| 3 | * | 57 | 3 | ⊕ | | 3 | + | 133 |
| 4 | Q | 68 | | | | 4 | P | 144 |
| 5 | ⊕ | | | | | 5 | ⊙ | |

PREC(+) < PREC(*)
$Q*R \rightarrow R_1$

PREC(+) > PREC(")")

$S_{\beta-1}$ is an operator
∴ $-P \rightarrow R_2$

| $\alpha$ | $L_\alpha$ | $S_\beta \rightarrow L_\alpha$ | $\alpha$ | $L_\alpha$ | $S_\beta \rightarrow L_\alpha$ | $\alpha$ | $L_\alpha$ | $S_\beta \rightarrow L_\beta$ |
|---|---|---|---|---|---|---|---|---|
| 0 | ⊣ | 20 | 0 | ⊣ | 20 | 0 | ⊣ | 20 |
| 1 | ) | 31 | 1 | ) | 31 | 1 | $R_3$ | 260 |
| 2 | $R_1$ | 101 | 2 | $R_3$ | 226 | 2 | / | 275 |
| 3 | + | 133 | 3 | ① | | 3 | ) | 286 |
| 4 | $R_2$ | 174 | | | | 4 | M | 297 |
| 5 | ① | | | | | 5 | / | 312 |
| | | | | | | 6 | L | 323 |
| | | | | | | 7 | ⊙ | |

PREC(")") < PREC("+")
$R_2 + R_1 \rightarrow R_3$

$L_{\alpha-1} \rightarrow L_{\alpha-2}$

PREC(-) ≠ -u < PREC(/)
$L/M \rightarrow R_4$

## Status of LList at Various Intervals (Continued)

| $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ |
|---|---|---|
| 0 | ⊣ | 20 |
| 1 | $R_3$ | 260 |
| 2 | / | 275 |
| 3 | ) | 286 |
| 4 | $R_4$ | 363 |
| 5 | - | 395 |
| 6 | K | 406 |
| 7 | ① | |

$S_{\beta-2} \neq$ .ABS.
$\therefore$ J↑K → $R_5$

| $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ |
|---|---|---|
| 0 | ⊣ | 20 |
| 1 | $R_3$ | 260 |
| 2 | / | 275 |
| 3 | ) | 286 |
| 4 | $R_4$ | 363 |
| 5 | - | 395 |
| 6 | $R_5$ | 440 |
| 7 | ① | |

PREC(")") < PREC(-)
$\therefore R_5 - R_4 \to R_6$

| $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ |
|---|---|---|
| 0 | ⊣ | 20 |
| 1 | $R_3$ | 260 |
| 2 | / | 275 |
| 3 | ) | 286 |
| 4 | $R_6$ | 493 |
| 5 | ① | |

$(R_i)$ CASE
$\therefore L_{\alpha-1} \to L_{\alpha-2}$

| $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ |
|---|---|---|
| 0 | ⊣ | 20 |
| 1 | $R_3$ | 260 |
| 2 | / | 275 |
| 3 | $R_6$ | 527 |
| 4 | ⨁ | |

PREC(+) < PREC(/)
$\therefore R_6/R_3 \to R_2$

| $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ |
|---|---|---|
| 0 | ⊣ | 20 |
| 1 | $R_7$ | 560 |
| 2 | + | 592 |
| 3 | ) | 603 |
| 4 | I | 614 |
| 5 | .ABS, | |

.ABS. I → $R_8$

| $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ |
|---|---|---|
| 0 | ⊣ | 20 |
| 1 | $R_7$ | 560 |
| 2 | + | 592 |
| 3 | ) | 603 |
| 4 | $R_8$ | 637 |
| 5 | + | 671 |
| 6 | F | 682 |
| 7 | ↑ | |

$S_{\beta-2} \neq$ .ABS.
$\therefore$ E↑F → $R_9$

## Status of LList at Various Intervals (Continued)

| $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ | $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ | $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ |
|---|---|---|---|---|---|---|---|---|
| 0 | ⊣ | 20 | 0 | ⊣ | 20 | 0 | ⊣ | 20 |
| 1 | $R_7$ | 560 | 1 | $R_7$ | 560 | 1 | $R_7$ | 560 |
| 2 | + | 592 | 2 | + | 592 | 2 | + | 592 |
| 3 | ) | 603 | 3 | ) | 603 | 3 | $R_{10}$ | 803 |
| 4 | $R_8$ | 637 | 4 | $R_{10}$ | 769 | 4 | * | 818 |
| 5 | + | 671 | 5 | ⓘ | | 5 | ) | 829 |
| 6 | $R_9$ | 716 | | | | 6 | D | 840 |
| 7 | ⓘ | | | | | 7 | * | 855 |
| | | | | | | 8 | C | 866 |
| | | | | | | 9 | ⊥ | |

PREC("(") < PREC(+)  
∴ $R_9 + R_8 \to R_{10}$

$(R_1)$CASE  
∴ $L_{\alpha-1} \to L_{\alpha-2}$

PREC(+) < PREC(*)  
∴ $C*D \to R_{11}$

| $\alpha$ | $L_\alpha$ | $S_\beta \quad L_\alpha$ | $\alpha$ | $L_\alpha$ | $S_\beta \quad L_\alpha$ | $\alpha$ | $L_\alpha$ | $S_\beta \quad L_\alpha$ |
|---|---|---|---|---|---|---|---|---|
| 0 | ⊣ | 20 | 0 | ⊣ | 20 | 0 | ⊣ | 20 |
| 1 | $R_7$ | 560 | 1 | $R_7$ | 560 | 1 | $R_7$ | 560 |
| 2 | + | 592 | 2 | + | 592 | 2 | + | 592 |
| 3 | $R_{10}$ | 803 | 3 | $R_{10}$ | 803 | 3 | $R_{10}$ | 803 |
| 4 | * | 818 | 4 | * | 818 | 4 | * | 818 |
| 5 | ) | 829 | 5 | ) | 829 | 5 | ) | 829 |
| 6 | $R_{11}$ | 899 | 6 | $R_{11}$ | 899 | 6 | $R_{13}$ | 1051 |
| 7 | * | 931 | 7 | + | 931 | 7 | ⓘ | |
| 8 | B | 942 | 8 | $R_{12}$ | 1001 | | | |
| 9 | * | 957 | 9 | ⓘ | | | | |
| 10 | A | 968 | | | | | | |
| 11 | ( | | | | | | | |

PREC("(") < PREC(*)  
∴ $A*B \to R_{12}$

PREC("(") < PREC(+)  
$R_{12} + R_{11} \to R_{13}$

$(R_2)$CASE  
$L_{\alpha-1} \to L_{\alpha-2}$

-319-

## Status of LList at Various Intervals (Continued)

| $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ | $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ | $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ |
|---|---|---|---|---|---|---|---|---|
| 0 | ⊣ | 20 | 0 | ⊣ | 20 | 0 | ⊣ | 20 |
| 1 | $R_7$ | 560 | 1 | $R_7$ | 560 | 1 | $R_{15}$ | 1168 |
| 2 | + | 592 | 2 | + | 592 | 2 | = | 1200 |
| 3 | $R_{10}$ | 803 | 3 | $R_{14}$ | 1118 | 3 | Z | 1211 |
| 4 | * | 818 | 4 | ⊜ | | 4 | ⊕ | |
| 5 | $R_{13}$ | 1085 | | | | | | |
| 6 | ⊜ | | | | | | | |

PREC(=) < PREC(*)          PREC(=) < PREC(+)          "TERM" CASE
$R_{13} * R_{10} \to R_{14}$       $R_{14} + R_7 \to R_{15}$       $Z = R_{15} \to R_{16}$

| $\alpha$ | $L_\alpha$ | $S_\beta \to L_\alpha$ |
|---|---|---|
| 0 | ⊣ | 1270 |
| 1 | $R_{16}$ | 1244 |
| 2 | ⊕ | |

## Object Program

| | MNEMONIC | TAG | ADDRESS | | |
|---|---|---|---|---|---|
| MLIST | LDQ | | POOL + 14 | 91 | $C(Q) \to MQ$ |
| | FMP | | POOL + 15 | 97 | $C(Q) * C(R) \to MQ$ |
| | STQ | | POOL + 100 | 107 | $MQ \to R_1$ |
| | CLA | | POOL | 164 | $ZERO \to AC$ |
| | FSB | | POOL + 13 | 170 | $C(AC) - C(P) \to AC$ |
| | STO | | POOL + 101 | 180 | $AC \to R_2$ |
| | CLA | | POOL + 101 | 216 | $C(R_2) \to AC$ |
| | FAD | | POOL + 100 | 222 | $C(R_2) + C(R_1) \to AC$ |
| | STO | | POOL + 102 | 232 | $C(AC) \to R_3$ |
| | CLA | | POOL + 11 | 353 | $C(L) \to AC$ |

## Object Program (Continued)

| | | | |
|---|---|---|---|
| FDP | | POOL + 12 | 359 $C(L)/C(M) \rightarrow MQ$ |
| STQ | | POOL + 103 | 369 $C(Q) \rightarrow R_4$ |
| LDQ | | POOL + 9 | 426 $C(J) \rightarrow MQ$ |
| CLA | | POOL + 10 | 432 $C(K) \rightarrow AC$ |
| TSX | 1 | EXP | 436 GO TO SUBROUTINE "EXP" |
| STQ | | POOL + 104 | 446 RETURN HERE $C(X1) + 1$ |
| CLA | | POOL + 104 | 483 $C(R_5) \rightarrow AC$ |
| FSB | | POOL + 103 | 489 $C(R_5) - C(R_4) \rightarrow AC$ |
| STO | | POOL + 105 | 499 $C(AC) \rightarrow R_6$ |
| CLA | | POOL + 105 | 550 $C(R_6) \rightarrow AC$ |
| FDP | | POOL + 102 | 556 $C(R_6)/C(R_3) \rightarrow MQ$ |
| STQ | | POOL + 106 | 566 $C(MQ) \rightarrow R_7$ |
| CLA | | POOL | 627 ZERO $\rightarrow AC$ |
| FAM | | POOL + 8 | 633 $|C(I)| \rightarrow AC$ |
| STO | | POOL + 107 | 643 $AC \rightarrow R_8$ |
| LDQ | | POOL + 6 | 702 $C(E) \rightarrow MQ$ |
| CLA | | POOL + 7 | 708 $C(F) \rightarrow AC$ |
| TSX | 1 | EXP | 712 GO TO SUBROUTINE "EXP" |
| STQ | | POOL + 108 | 722 RETURN $C(MQ) \rightarrow R_9$ |
| CLA | | POOL + 108 | 759 $C(R_9) \rightarrow AC$ |
| FAD | | POOL + 107 | 765 $C(R_9) + C(R_8) \rightarrow AC$ |
| STO | | POOL + 109 | 775 $C(AC) \rightarrow R_{10}$ |
| LDQ | | POOL + 4 | 889 $C(C) \rightarrow MQ$ |
| FMP | | POOL + 5 | 895 $C(C)*C(D) \rightarrow MQ$ |
| STQ | | POOL + 110 | 905 $MQ \rightarrow R_{11}$ |
| LDQ | | POOL + 2 | 991 $C(A) \rightarrow MQ$ |
| FMP | | POOL + 3 | 997 $C(A)*C(B) \rightarrow MQ$ |
| STQ | | POOL + 111 | 1007 $MQ \rightarrow R_{12}$ |
| CLA | | POOL + 111 | 1011 $R_{12} \rightarrow AC$ |

#### Object Program (Continued)

| MLIST | FAD | POOL + 110 | 1047 | $C(R_{12}) + C(R_{11}) \rightarrow AC$ |
|---|---|---|---|---|
| (cont) | STO | POOL + 112 | 1057 | $AC \rightarrow R_{13}$ |
| | LDQ | POOL + 112 | 1108 | $C(R_{13}) \rightarrow MQ$ |
| | FMP | POOL + 109 | 1114 | $C(R_{13}) * C(R_{10}) \rightarrow MQ$ |
| | STQ | POOL + 113 | 1124 | $MQ \rightarrow R_{14}$ |
| | CLA | POOL + 113 | 1158 | $C(R_{14}) \rightarrow AC$ |
| | FAD | POOL + 106 | 1164 | $C(R_{14}) + C(R_7) \rightarrow AC$ |
| | STO | POOL + 114 | 1174 | $AC \rightarrow R_{15}$ |
| | CLA | POOL + 114 | 1234 | $C(R_{15}) \rightarrow AC$ |
| | STO | POOL + 1 | 1240 | $C(AC) \rightarrow Z$ |
| | NOP | POOL + 115 | 1250 | $R_f + 1$ |

| POOL | 00 | 00 | 00 | 00 | 00 | 00 | ZERO |
|---|---|---|---|---|---|---|---|
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for Z |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for A |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for B |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for C |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for D |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for E |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for F |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for I |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for J |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for K |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for L |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for M |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for P |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for Q |
| | 00 | 00 | 00 | 00 | 00 | 00 | Reserved for R |

| POOL + 100 | 00 | 00 | 00 | 00 | 00 | 00 | Result of | $Q*R = R_1$ |
|---|---|---|---|---|---|---|---|---|
| | 00 | 00 | 00 | 00 | 00 | 00 | | $\phi - P = R_2$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $R_2 + R_1 = R_3$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $L/M = R_4$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $J\uparrow K = R_5$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $R_5 - R_4 = R_6$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $R_6/R_3 = R_7$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $.ABS. \ I = R_8$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $E\uparrow F = R_9$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $R_9 + R_8 = R_{10}$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $C*D = R_{11}$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $A*B = R_{12}$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $R_{12} + R_{11} = R_{13}$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $R_{13}*R_{10} = R_{14}$ |
| | 00 | 00 | 00 | 00 | 00 | 00 | | $R_{14} + R_7 = R_{15}$ |

## 5. CONCLUSIONS

When comparing the compilation time of the simplified expressions to that of its equivalent complex expression, it is readily seen that the writing of complex expressions is to a marked advantage. For most compilers, however, this is not the case and the programmer can usually compile simplified expressions with greater speed. The compiler size could be reduced considerably if only the simplified type of expressions is to be compiled but, of course, this would mean a sacrifice of compilation time. If the generated object program (Appendix D) is examined, it is readily seen that this is not an optimum object program. To generate an optimum object code, additional tests could be included in the compiler but these changes would increase compilation time.

Many of today's compilers are designed with the desire of generating optimum object program in mind. In order to accomplish this, the substitution expressions are scanned several times looking for:

1. Repetitive triples, i.e.:

   $x = a + b$,

   $y = C + D*(a + b)$,

   where $a + b$ is the repetitive triple

2. Commutative triples, i.e.:

   $x = a + b$,

   $y = C + D*(b + a)$,

   where $a + b = b + a$, etc.

3. Redundant object code, i.e.:

   STO $R_i$        $AC \longrightarrow R_i$

   CLA $R_i$        $R_i \longrightarrow AC$

In the present algorithm, none of the above considerations was employed. For large programs and programs that are developed with long-range use in mind, the above features should be considered in the writing of a compiler. Such a compiler would be useful in a production-type computer environment. Such features sometimes would result in long compilation times to accomplish short object program execution times. If the object program is to be used only once or twice, it sometimes becomes absurd to use as much as twice the object program execution time in order to accomplish a compilation.

The accumulative compilation time for compiling the expression

$$Z = (A*B + C*D)*(E\uparrow F + .ABS. I) + (J\uparrow K - L/M)/(-P + Q*R)$$

was found to be 2.37 msec and for an equivalent set of simplified expressions (using a more general subroutine), 3.27 msec. The algorithm consists of 184, IBM 7090 instructions and 654 locations for constants and working storage.

## APPENDIX XIII - MACHINE II PROGRAMMING

### 1. INTRODUCTION

This is the programming report for the compilation problem programmed
for Machine II as a portion of the work performed under Contract AF30-
(602)-3550. The compilation problem was the programming of a portion
of the Michigan Algorithm Decoder (MAD). The section of MAD chosen
for demonstration was the compilation of substitution statements. Sub-
stitution statements are composed of variables and operators whose values
are substituted or made equal to some variable.

When grouped, the elements of the statement fall into sets of triples, two
operands and an operator, that can be used to generate an object program
for machine execution. The keys to statement decomposition and object
generation are operator precedence, the order of execution when a state-
ment is composed of various operators, and a statement scanner.

Item 2 contains a discussion of the programs. Item 3 contains results of
the programming, a comparison with the IBM 7090 program, and the com-
piler flow charts and programs. Item 4 contains a discussion of the ob-
ject program generated and the object program.

### 2. DISCUSSION OF THE PROGRAM

#### a. Statement

The statement selected for demonstration is

$$Z = (A \cdot B + C \cdot D) + (E \uparrow F + ABS \ I) +$$

$$(J \uparrow K - L/M)/(NEG \ P + Q \cdot R)$$

Each variable is represented as a single alphabetic character, although a maximum of six are recognized in the MAD translator. It is assumed that the decoding process has been completed and a table (the L list) generated with single-word entries corresponding to each variable and each operator in the statement. The entries in the L list are in the same order as the elements in the substitution statement. The L list is scanned from left to right and, depending on the tests that are satisfied, an output list (the P list) and a temporary list (S list, or stack) are generated.

Operands are transferred immediately from the L list to the P list. Operators are transferred to the top of the S list if their precedence is equal to or greater than the current operator on the top of this list. Termination and grouping symbols require special handling. Left parentheses are unconditionally put on the S list. Right parentheses cause the removal of all elements from the S list and transfer to the P list with the parentheses, both right and left, then removed. The right termination symbol causes the transfer of all remaining operators in the S list to the P list.

The substitution statement is assumed to have less than 256 elements so that it can be considered as occupying at most one block of memory.

The object program also is assumed to require at most one block of memory. These restrictions are not necessarily fixed but could be removed with minor programming changes.

When an operator is added to the P list, the preceding two operands with the operator are sent to a generator program to produce a segment of the object program. The resultant, $R_i$, is then entered in the P list and the scanning continued.

Table XIII-1 shows the L list. Table XIII-2 shows the P list status and the compiled triples.

## TABLE XIII-1 - L LIST

| Element (no.) | Element | Element (no.) | Element | Element (no.) | Element |
|---|---|---|---|---|---|
| 1 | Z | 14 | E | 27 | L |
| 2 | = | 15 | ↑ | 28 | / |
| 3 | ( | 16 | F | 29 | M |
| 4 | A | 17 | ÷ | 30 | ) |
| 5 | • | 18 | ABS | 31 | / |
| 6 | B | 19 | I | 32 | ( |
| 7 | + | 20 | ) | 33 | NEG |
| 8 | C | 21 | + | 34 | P |
| 9 | • | 22 | ( | 35 | + |
| 10 | D | 23 | ' | 36 | Q |
| 11 | ) | 24 | ; | 37 | • |
| 12 | • | 25 | K | 38 | R |
| 13 | ( | 26 | - | 39 | ) |

## TABLE XIII-2 - P LIST STATUS AND

## COMPILED TRIPLES

| P list status | | | | | | Triples compiled |
|---|---|---|---|---|---|---|
| Z | A | B | • | | | $R_5 = A \cdot B$ |
| Z | $R_5$ | C | D | • | | $R_9 = C \cdot D$ |
| Z | $R_5$ | $R_9$ | + | | | $R_7 = R_5 + R_9$ |
| Z | $R_7$ | E | F | | | $R_{15} = E \uparrow F$ |
| Z | $R_7$ | $R_{15}$ | ! | ABS | | $R_{18} = ABS\ I$ |
| Z | $R_7$ | $R_{15}$ | $R_{18}$ | | | $R_{17} = R_{15} + R_{18}$ |
| Z | $R_7$ | $R_{17}$ | • | | | $R_{12} = R_7 \cdot R_{17}$ |
| Z | $R_{12}$ | J | K | ↑ | | $R_{24} = J \uparrow K$ |
| Z | $R_{12}$ | $R_{24}$ | L | M | / | $R_{28} = L/M$ |
| Z | $R_{12}$ | $R_{24}$ | $R_{28}$ | • | | $R_{26} = R_{24} \cdot R_{28}$ |
| Z | $R_{12}$ | $R_{26}$ | P | NEG | | $R_{33} = NEG\ P$ |
| Z | $R_{12}$ | $R_{26}$ | $R_{33}$ | Q | P • | $R_{37} = Q \cdot R$ |
| Z | $R_{12}$ | $R_{26}$ | $R_{33}$ | $R_{37}$ | • | $R_{35} = R_{13} \cdot R_{37}$ |
| Z | $R_{12}$ | $R_{26}$ | $R_{35}$ | / | | $R_{31} = R_{26} / R_{35}$ |
| Z | $R_{12}$ | $R_{31}$ | • | | | $R_{21} = R_{12} \cdot R_{31}$ |
| Z | $R_{21}$ | • | | | | |
| • | | | | | | |

A flow chart depicting the process of starting a generator program
is shown in Figure XIII-1 (the illustrations begin on Page 335). The
* routines have found that an operator is to be transferred to the P
list. When the transfer has been accomplished, an instruction starts
the SP block; this in turn starts either SPU or SPB. When the required
data have been transferred to the generator programs, G1, G2, G3, or
G4 BRING instructions are executed in SP and G*, I*, or K*, enabling
starting of subsequent blocks of the program. The BRING instructions
request a result that is generated in a block started by the block in
which the BRING resides. The BRING is executed when the result is
generated. Hence, a block is not completely executed until the
BRING is executed. If subsequent STARTs are dependent on a BRING,
there will be a delay in them until execution of the BRING.

b. Assumptions

A variable in the L list or P list has the format shown below.

| i | INDEX | V | ADDRESS |
|---|-------|---|---------|

The address of the whole word as it is contained in a block is i. Index
is a pointer to a list location containing the symbolic name of the vari-
able. V is a bit, equal to a one, indicating that the word corresponds
to a variable. Address is the memory address of the variable that is
assumed as fixed after L list generation.

An operator in the L, S, or P list has the format shown below.

| i | PREC | V | CODE |
|---|------|---|------|

The address of the whole word as it appears in a block is i. The
precedence of the operator is contained in the PREC field. V is a
bit equal to 0, indicating that the element is an operator. Code is
the operation code.

The operations considered were floating point arithmetic, absolute value, exponentiation, and equality.

c. Program Techniques

The program is a left-to-right statement scanner that allows generation of triples as soon as an operation can be transferred from the S list to the P list. In the flow diagram of Figure XIII-2, the ABLE and CHARLIE loop transfers operands from the L list to the P list. If an element is an operator and its precedence is equal to or greater than the top element of the S list, the S list is pushed down and the element is put on top of the S list. If the operator has less precedence than the top element of the S list, then the top element of the S list is put on the P list. When an operator is placed on the P list, a generator program is started that produces the corresponding portion of the object program using this operator and the two preceding operands on the P list. A resultant address is calculated and entered in the P list as a variable.

The detection of a left parenthesis results in the transferral of the element to the top of the stack. A right parenthesis causes all operators up to the first left parenthesis on the S list to be transferred, in order, to the P list. Again, each operator transferred to the P list has a corresponding generator program started that produces a portion of the object program corresponding to the operator transferred. The parentheses are then dropped from the S list and are not transferred to the P list.

When a right termination is detected, all remaining operators on the S list are transferred to the P list in order. For each operator transferred, a corresponding generator program is started to produce a portion of the object program.

A triple is detected each time an operator is transferred from the S list to the P list. The operands associated with the operator and

comprising the triple are immediately ahead of the operator on the
P list. The resultant address is calculated as the first word address
of the generated object program segment. This resultant address is
then placed in the P list as a variable. It will in turn be considered
as an element of some triple.

The generator programs produce a segment of the object program.
The resultant is always left in the first word of the object program
segment or, in the case of a double precision operation, in the first
two words of the segment. The initial address of a segment is cal-
culated easily by considering the maximum size of all possible object
program segments. In this example, the size is indicated in the pro-
gram as $\bigotimes$ . The index of the operator in the L list is used to de-
termine i $\bigotimes$ , which is then added to the object program base address
and results in OBI + I $\bigotimes$ , the initial address of the object program
segment and the location of the resultant of the triple.

Parallelism exists both within any block of the program and between
blocks of the program and the generator programs. The algorithm
is sequential in nature; attempts to scan the substitution statement
in parallel results in invalid intermediate forms of the statement.
Attempts to correct this uncovered a processor control problem.
Figure XIII-1 depicts the sequence of a program started when an
operator is detected. The solid lines indicate the sequence executed
for a block start. The dashed lines indicate the path followed while
the initiating block is waiting for an answer from the initiated block.
The initiated block in turn tests the operator and starts other blocks.

The G* G' loop in Figure XIII-1 starts the SP block which tests the
operator to determine if it is a unary or binary operator. If it is
unary, the SPU block is started and a return to the G is accomplished
by the BRING instruction in G. If the operator is binary, the SPB
block is started. The SPB block tests the operator and if it is not
an exponentiation, the GENO 1 block is started; if it is exponentiation,

-330-

the GENO 3 is started. The SPB block is tested also by the BRING instruction in G and when it is satisfied the GJ loop is reinitiated.

Some characteristics of Machine II make it a desirable machine. The storage of results in the word occupied by the generating instruction simplifies programming in that no explicit store instruction is required for MPC storage. However, a store instruction is required for a memory store operation. The variety of conditional starts is necessary to enable branching from a block. To execute a two-way branch from a block, two conditionals are required in that block, both testing the same word. In some respects, this is undesirable in that extra instructions are required over what would be required in a sequential machine. However, no extra time is required because the two tests are performed in parallel as soon as the test word is available.

The ability to acquire data from neighboring blocks is necessary for program continuity. Data from a previous block can be addressed relative to the previous starting instruction or by absolute block address. Data from a subsequent block can be retrieved by absolute address within the started block. A BRING and an M WAIT instruction for a block to be started should not be used in the same block. The M WAIT instruction waits until all instructions have been executed before starting a new block while the BRING attempts to bring back a piece of data generated by that block. However, a conditional or an unconditional start and a BRING instruction are compatible.

Data in memory can be found with the READ MEMORY, READ MEMORY INDIRECT, and THRESHOLD instructions. The READ MEMORY is an absolute address instruction, the READ INDIRECT permits an indexed access, and the THRESHOLD instruction permits a next-higher-than threshold search.

A disadvantage is the quantity of data that must be passed from one block to another. In the problem, as many as 8 to 10 words passed

through a series of blocks. Timewise, this was not detrimental since all the data were passed in parallel. However, each transfer required an instruction. In some cases, where a word was being used as an index, the shift instruction is placed elsewhere in the program and the index augmented with the result left in its proper relative location ready for transferral to the next block. Considerable time is spent in laying out data transfers, especially when attempting to implement a program loop.

The instruction-erase option allows the programmer to maintain a minimum of MPC storage. As results are used, they can be erased, allowing MPC storage of other data. In the problem, however, it was found that because of the amount of branching and looping the task of erasing data or maintaining data and letting subsequent blocks erase it where there was alternate subsequent blocks, became difficult and time consuming. One way to surmount the problem would be to alter the erase instructions so that the instruction would not be executed until all data had been transferred to the new block. Then the previous block could be wiped out completely. The instruction would be a combination of ERASE and a modified M WAIT where the wait would be dependent on the transferral of all data requested by the new block of program.

Programming of Machine II has been relatively easy in some respects and difficult in others. It is easy to program in a straightforward manner with no attempt at finesse. However, the lack of indexing seriously hampers any attempt to reduce the size of a program by using loops. The ability to converse between blocks using the SHIFT and BRING instructions partly resolves the difficulty, but the necessity for time consuming and tedious program layout of data to be transferred between blocks still remains when operation is restricted to the MPC. It is possible that even this difficulty can be removed by maintaining these data in memory, but here again

the tradeoff sacrifices the speed of obtaining a result from the MPC
for obtaining a result from the memory. The former is obtained
while executing an instruction requesting the result while the latter
requires a READ MEMORY instruction and then an operating instruc-
tion to acquire the same result.

3. RESULTS AND COMPARISON

The results looked for in the programming portion of the study are time
to execute the problem, processors used during execution, ease or dif-
ficulty in programming, comparison with the results of programming a
similar problem on a sequential machine, and an extrapolation of times
for the two machines and subsequent comparison.

The algorithm finally used in the problem was more sequential in nature
than others examined but did not display the control problems inherent in
some of the others when programmed. The result was a very low average
load on the processors. The average loading was 1.17 processors per
machine cycle for the total problem execution time. Peak loading is es-
timated to be no more than 10 processors in any machine cycle. Another
algorithm or a slightly longer or shorter program block could change the
peak, either increasing it or leveling it.

The time required to execute the algorithm for the translation of the sub-
stitution statement and generation of the corresponding object program
was 53.585 msec. The speed ratio of the sequential machine to Machine
II is N to 27, where N is the number of statements to be compiled. If it
is assumed that there is more than one statement to be translated, then
the sequential machine will translate faster until the number of statements
is 27. The fact that there are numerous processors and that the proces-
sor loading is small allows Machine II to process numerous statements in
parallel. When the statement loading exceeds 27, the sequential time re-
quired for translating will continue to increase linearly while the Machine
II time will remain relatively fixed at 54 msec. Assuming availability of

256 processors, then Machine II can average about $256/1.17 = 219$ statements every 54 msec. This would give Machine II a speed advantage over the IBM 7090 of approximately $219/27$ or 8 to 1. The actual time required for processing, of course, depends upon the sequence of operators and the amount of grouping within the substitution statement.

## 4. OBJECT PROGRAM

The object program generated is shown in Table XIII-4. Segments of the object program block are generated each time an operator is detected and transferred to the P list. Each segment is then stored in memory in the block assigned to the object program. Addresses within the block are determined by the index of the operand in the L list and the predetermined maximum size of any object program segment.

Some variables in the segments reside in memory and some reside in the MPC. Each generator determines where a variable is located and generates the appropriate instructions, either READ MEMORY or SHIFT THIS BLOCK. Upon execution, the variable replaces the instruction and the triple is executed with the resultant then occupying the first two words of the segment.

The object program can be executed in the same manner as the translator. It should be noted that the execution could be done simultaneously with translation so that the statement resultant would be available only shortly after object generation.
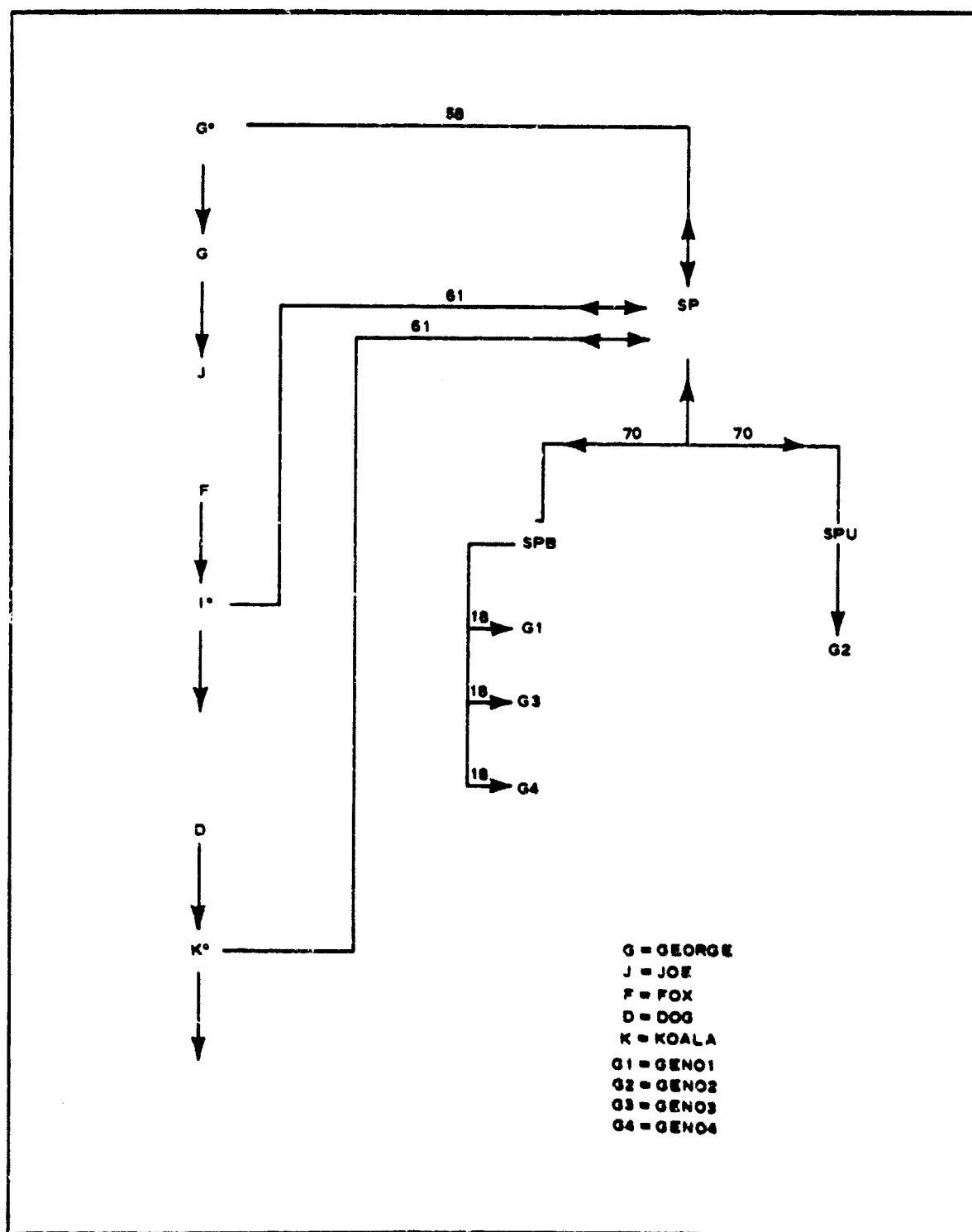
Figure XIII-1 - Flow Chart for Triple Generation Process
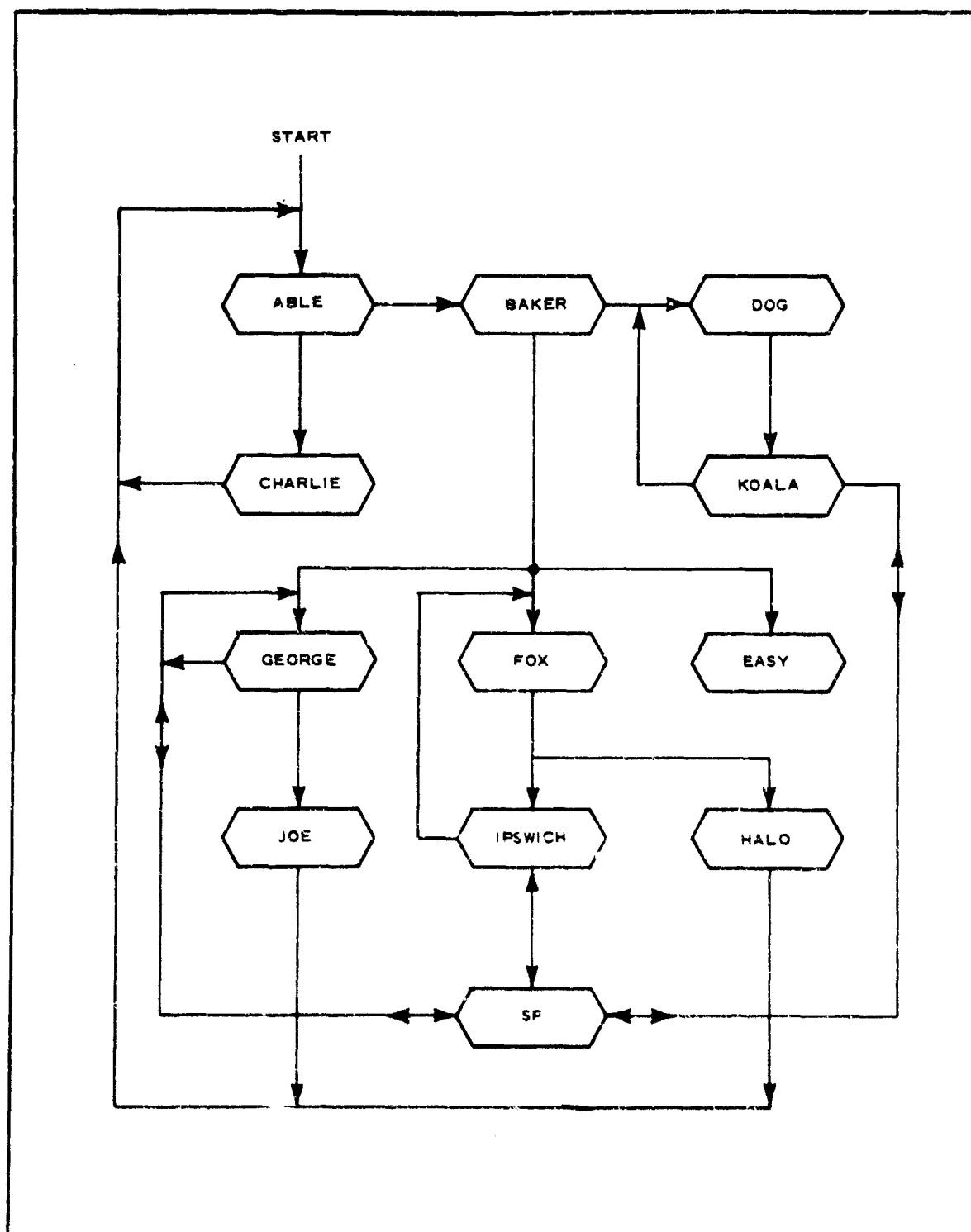
Figure XIII-2 - Master Flow Diagram
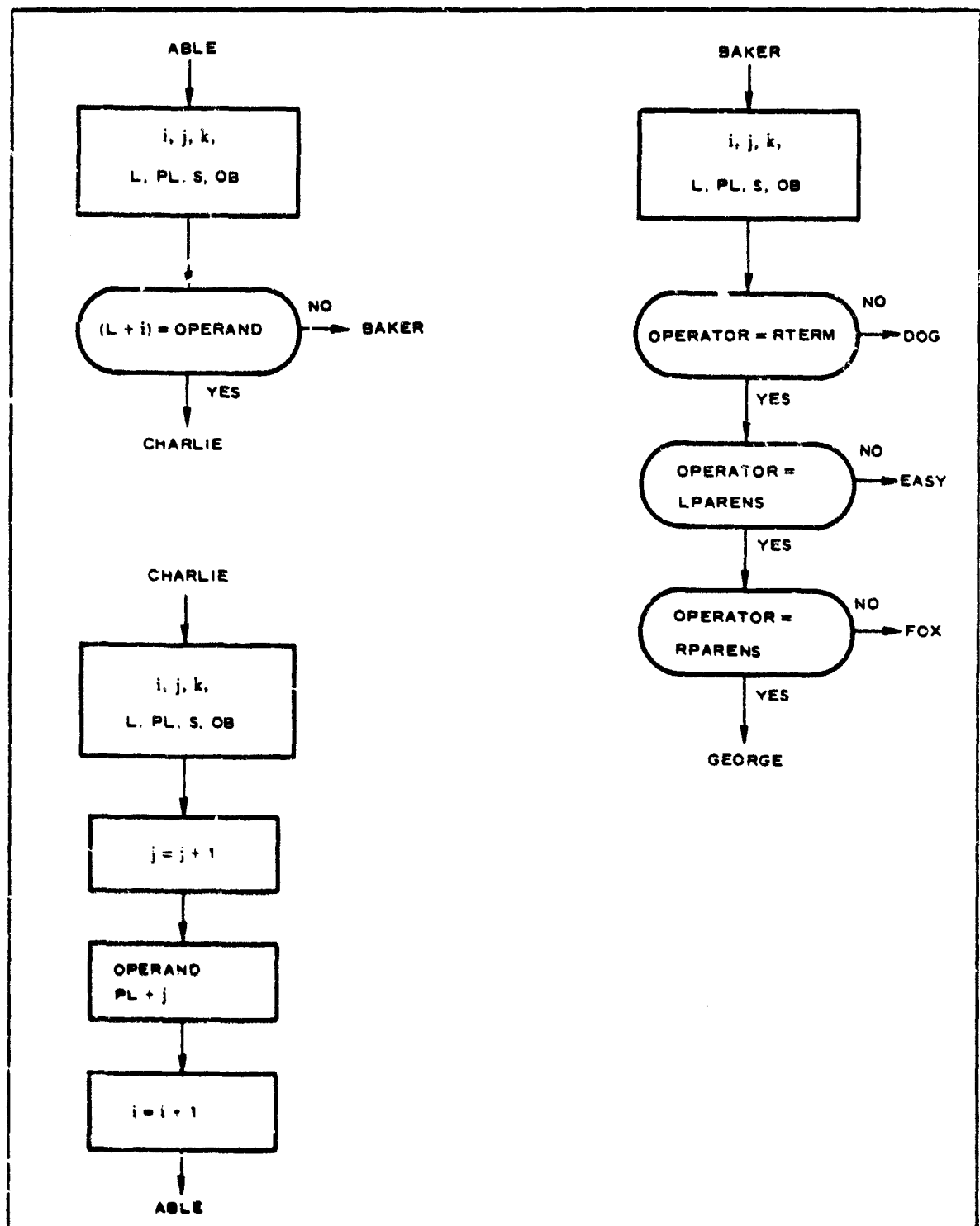
Figure XIII-3 - Able, Baker, Charlie Subroutines
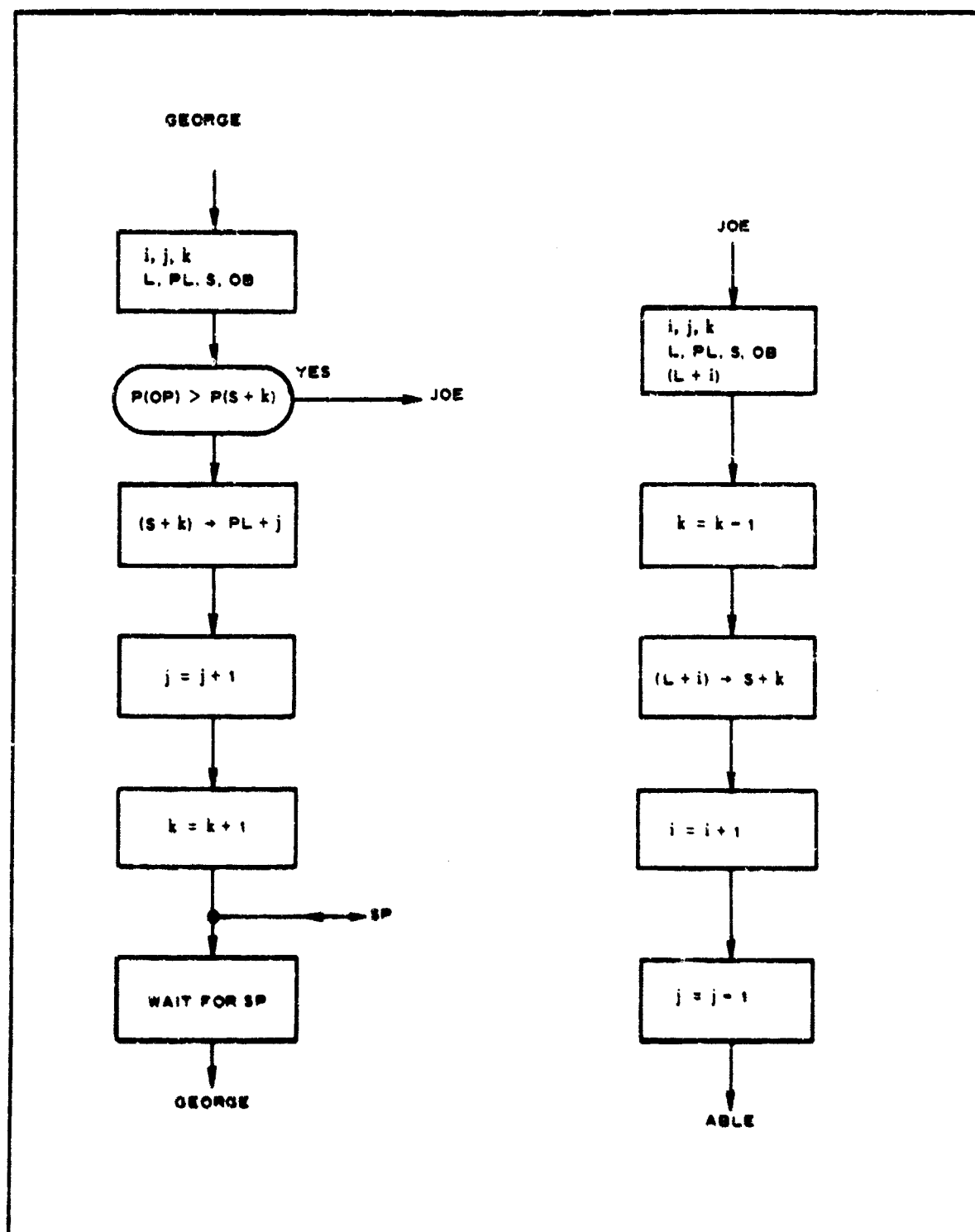
e XIII-4 - George, Joe Subroutines
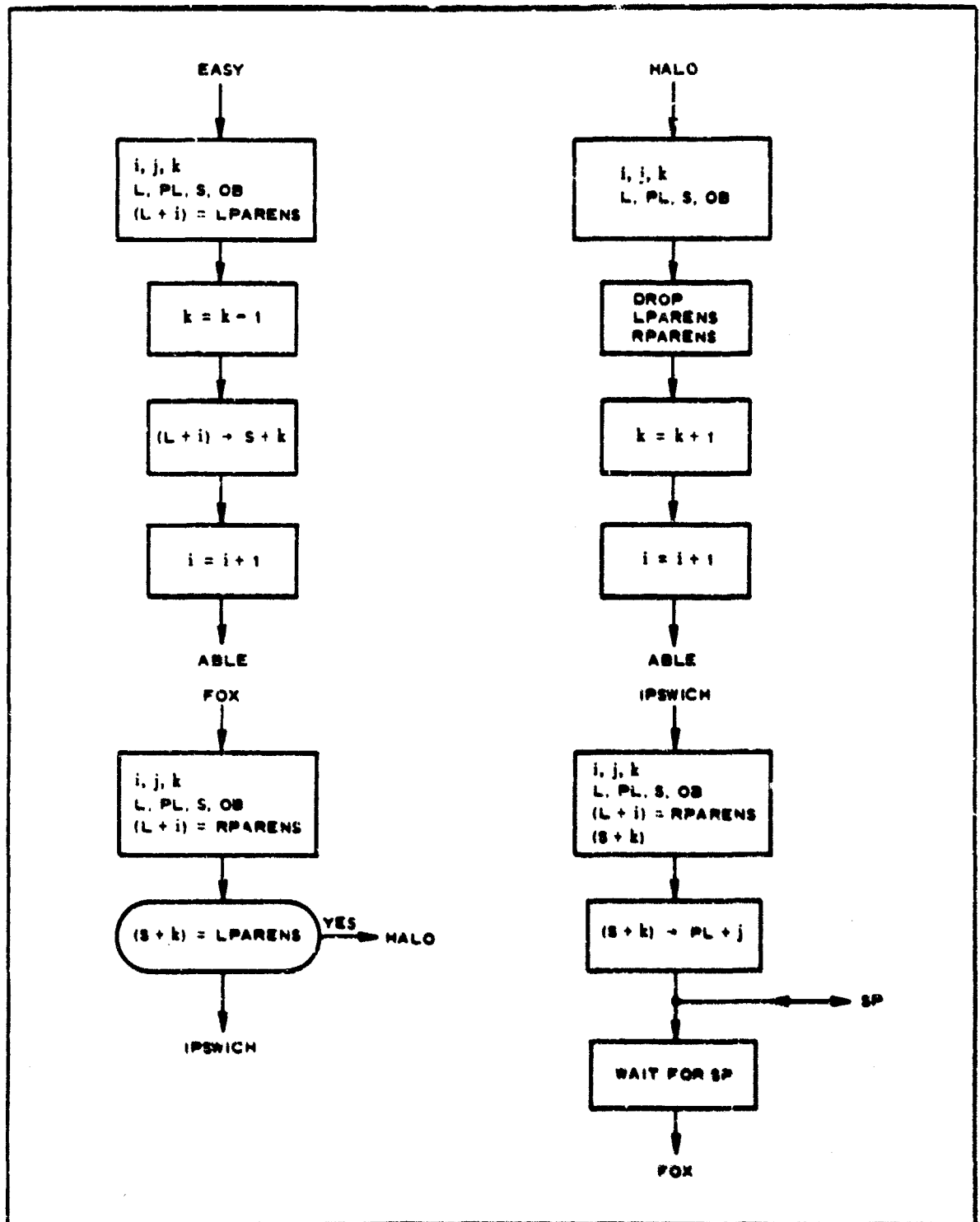
Figure XIII-5 - Easy, Fox, Halo, Ipswich Subroutines
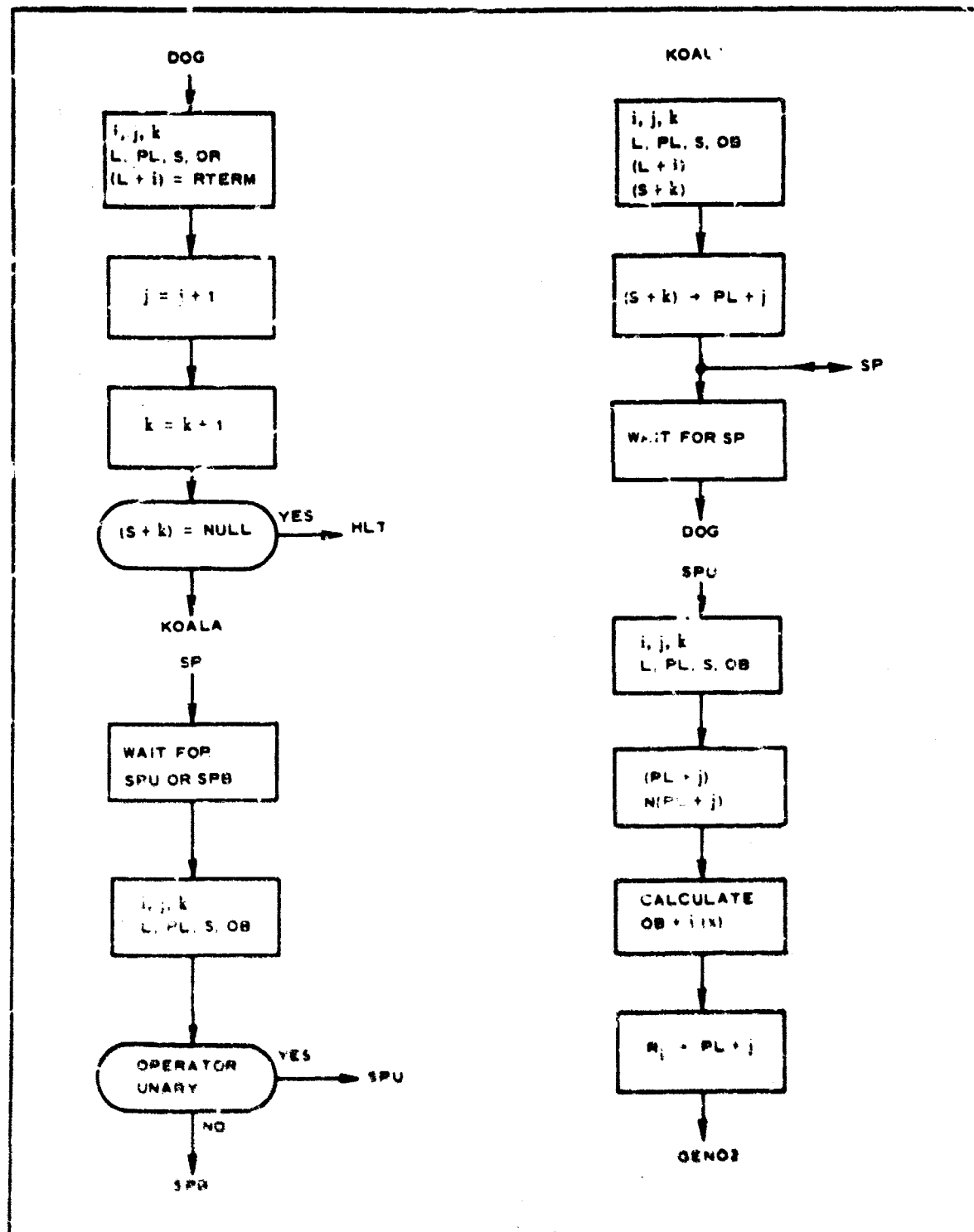
Figure XIII-6 - Dog. Koala, SP, SPU Subroutines
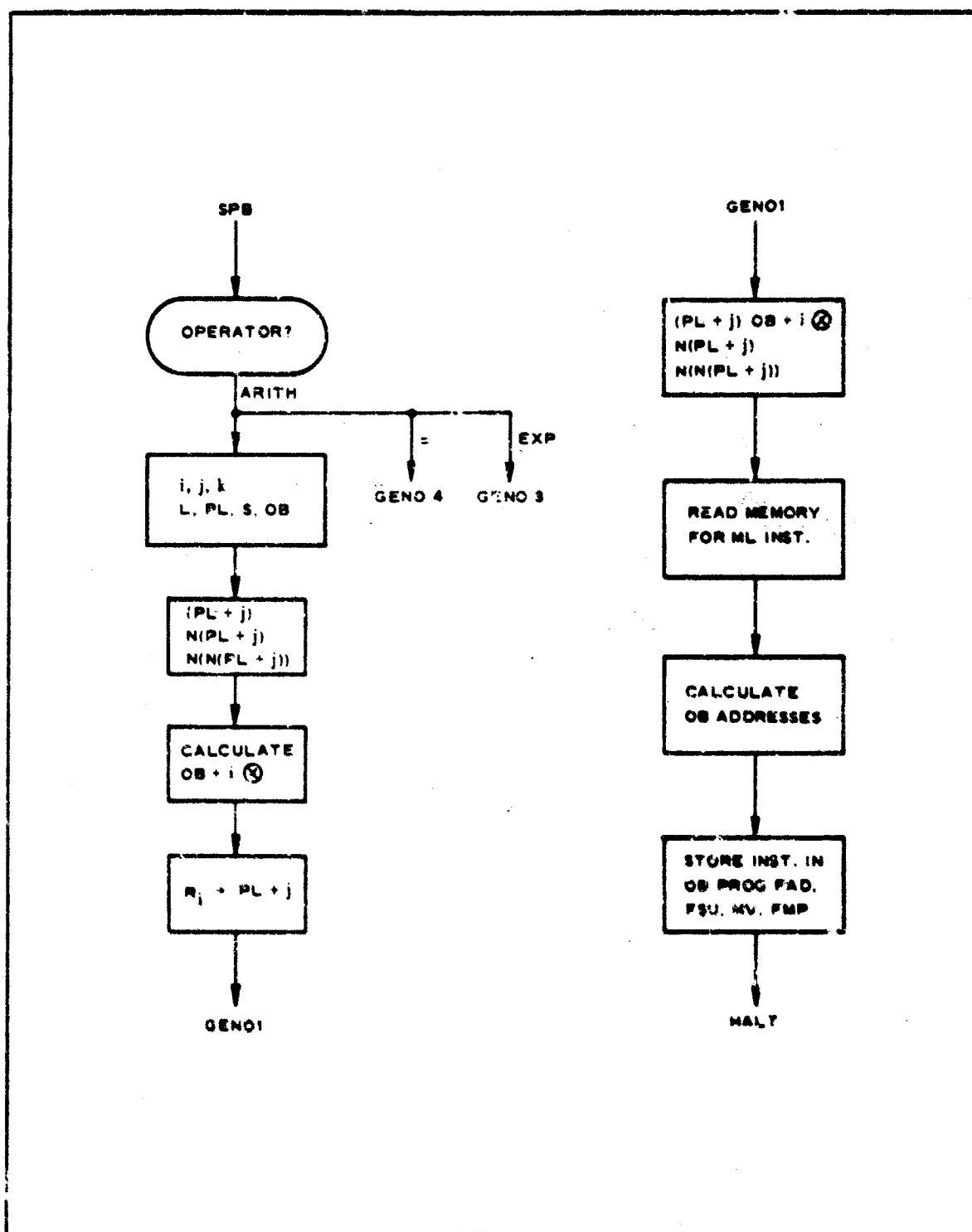
Figure XIII-7 - SPB, GENOI Subroutines
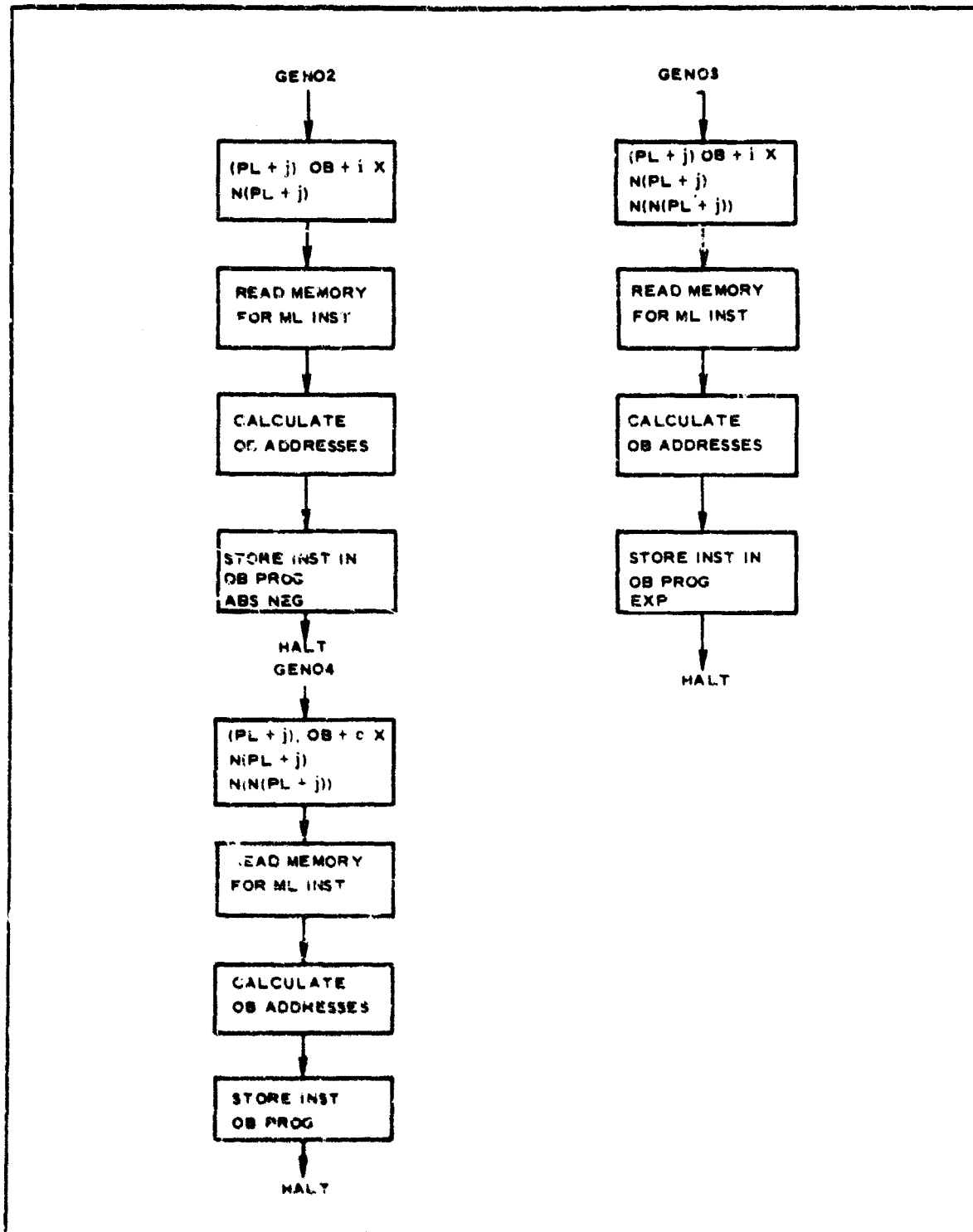
Figure XIII-8 - GENO2, GENO3, GENO4 Subroutines

## TABLE XIII-3 - COMPILER PROGRAM

| Item | | Instruction | | | Remarks | | Time P | Time R |
|------|------|------|------|------|---------|---|---|---|
| ABLE | α | SPB | 0 | 1 | i | a - - i | 6 | 7 |
| | +1 | SPB | 0 | 2 | j | a - - j | 6 | 7 |
| | +2 | SPB | 0 | 3 | k | a - - k | 6 | 7 |
| | β | SPB | 0 | 4 | L address a - L - | | 6 | 7 |
| | +1 | SPB | -0 | 5 | PL address a - PL - | | 6 | 7 |
| | +2 | SPB | 0 | 6 | S address a - S - | | 6 | 7 |
| | +3 | SPB | 0 | 7 | OB address a - OB - | | 6 | 7 |
| | γ + 1 | THS, M | α | β | (L + i) | | 10 | 11 |
| | +2 | LOI | γ + 1 | VBIT | | | 14 | 15 |
| | +3 | EQZ | γ + 2 | γ + 3 | | | 18 | |
| | +4 | CONS | | BAKER | OPERATOR | | - | 1 |
| | +5 | NEU | γ + 2 | γ + 6 | | | 18 | - |
| | +6 | CONS | | CHARLIE | OPERAND | | - | 11 |
| | +7 | CONS | | VBIT | | | - | 1 |
| BAKER | α | SPB | 0 | α | i | | 6 | 7 |
| | +1 | SPB | 0 | α + 1 | j | | 6 | 7 |
| | +2 | SPB | 0 | α + 2 | k | | 6 | 7 |
| | +3 | SPB | 0 | α + 3 | L | | 6 | 7 |
| | +4 | SPB | 0 | α + 4 | PL | | 6 | 7 |
| | +5 | SPB | 0 | α + 5 | S | | 6 | 7 |
| | +6 | SPB | 0 | α + 6 | OB | | 6 | 7 |
| | +7 | SPB | 0 | γ + 1 | (L + i) OPERATOR | | 6 | 7 |
| | β | LO6 | α + 7 | RTERM | A ● B | | 10 | 11 |
| | +1 | EQZ | β | β + 2 | 0 if RTERM, 1 otherwise | | 14 | - |
| | +2 | CONS | | DOG | | | - | 1 |
| | +3 | LO6 | α + 7 | LPARENS | A ● B | | 10 | 11 |
| | +4 | EQZ | β | β + 5 | 0 if LPARENS, 1 otherwise | | 14 | - |
| | +5 | CONS | | EASY | | | - | 1 |
| | +6 | LO6 | α + 7 | RPARENS | A ● B | | 10 | 11 |
| | +7 | EQZ | β | β + 8 | 0 if RPARENS, 1 otherwise | | 14 | - |
| | +8 | CONS | | FOX | | | - | 1 |
| | γ | MPY | β | β + 3 | | | 14 | 15 |
| | +1 | MPY | γ | β - 6 | | | 18 | 19 |
| | +2 | NEQ | γ + 1 | γ + 3 | if 0 then D = A + B + C | | 22 | |
| | +3 | CONS | | GEORGE | if 5 then D = ABC | | - | 1 |
| | +4 | CONS | | RTERM | | | - | 1 |
| | +5 | CONS | | LPARENS | | | - | 1 |
| | +6 | CONS | | RPARENS | | | - | 1 |
| CHARLIE | α | STB | 0 | γ + 1 | ( = i + 1 | | 11 | 8 |

NOTE: P is the cycle time after the block start in which a processor is active; R is the cycle time after the block start in which the result is available.

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | | Remarks | Time P | Time R |
|---|---|---|---|---|---|---|---|
| CHARLIE (cont) | +1 | STB | 0 | β + 1 | j = j + 1 | 17 | 18 |
| | +2 | SPB | 0 | α + 2 | k | 6 | 7 |
| | +3 | SPB | 0 | α + 3 | L | 6 | 7 |
| | +4 | SPB | 0 | α + 4 | PL | 6 | 7 |
| | +5 | SP3 | 0 | α + 5 | S | 6 | 7 |
| | +6 | SPB | 0 | α + 6 | OB | 6 | 7 |
| | β | SPB | 0 | α + 1 | j | 6 | 7 |
| | +1 | ADD | β | one | j = j + 1 - index address | 21 | 22 |
| | +2 | SPB | 0 | γ + 1 | (L + i) = OPND | 6 | 7 |
| | +3 | LO7 | α + 4 | β + 1 | PL + j - - PL j | 25 | 26 |
| | +4 | STO | β + 2 | β + 3 | | 29 | 30 |
| | γ | SPB | 0 | α | Get i | 6 | 7 |
| | +1 | ADD | γ | one | i = i + 1 | 10 | 11 |
| | +2 | MWT | | ABLE | | 32 | - |
| | +3 | CONS | | ONE | | | |
| DOG | α | SPB | 0 | α | i | 6 | 7 |
| | +1 | ADD | α + 8 | one | j    j = j + 1 | 10 | 11 |
| | +2 | ADD | α + 9 | one | k    k = k + 1 | 10 | 11 |
| | +3 | SPB | 0 | α + 3 | L | 6 | 7 |
| | +4 | SPB | 0 | α + 4 | PL | 6 | 7 |
| | +5 | SPB | 0 | α + 5 | S | 6 | 7 |
| | +6 | SPB | 0 | α + 6 | OB | 6 | 7 |
| | +7 | SPB | 0 | α + 7 | (L + i) = OPERATOR = RTERM | 6 | 7 |
| | +8 | SPB | 0 | α + 1 | j | 6 | 7 |
| | +9 | SPB | 0 | α + 2 | k | 6 | 7 |
| | +10 | THS | α + 9 | α + 5 | (s + k) | 10 | 11 |
| | +11 | LO6 | α + 10 | NULL | | 14 | 15 |
| | +12 | EQZ | α + 11 | α + 13 | | 18 | - |
| | +13 | CONS | | HALT | END | - | 1 |
| | +14 | NEQ | α + 11 | α + 15 | | 18 | |
| | +15 | CONS | | KOALA | | - | 1 |
| | +16 | CONS | | one | | - | 1 |
| | +17 | CONS | | null | | - | 1 |
| EASY | α | STB | 0 | α + 13 | i | 14 | 15 |
| | +1 | SPB | 0 | α + 1 | j | 6 | 7 |
| | +2 | STB | 0 | α + 16 | k | 14 | 15 |
| | +3 | SPB | 0 | α + 3 | L | 6 | 7 |
| | +4 | SPB | 0 | α + 6 | PL | 6 | 7 |
| | +5 | SPB | 0 | α + 5 | S | 6 | 7 |
| | +6 | SPB | 0 | α + 6 | OB | 6 | 7 |
| | +7 | SPB | 0 | α + 7 | (L + i) = OPERAND = LPARENS | 6 | 7 |
| | +8 | LO7 | α + 5 | α + 2 | s + k | 18 | 19 |

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | | Remarks | Time P | R |
|------|------|------|------|------|---------|---|---|
| EASY (cont) | +9 | STO | $a + 7$ | $a + 8$ | $(L + i) \to s + k - 1$ | 22 | 23 |
| | +10 | SPB | 0 | $a + 2$ | k | 6 | 7 |
| | +11 | SUB | $a + 10$ | one | $k = k - 1$ | 10 | 11 |
| | +12 | SPB | 0 | $a + 1$ | i | 6 | 7 |
| | +13 | ADD | $a + 12$ | one | $i = i + 1$ | 10 | 11 |
| | +14 | MWT | | ABLE | | 25 | - |
| | +15 | CONS | | ONE | | - | 1 |
| FOX | $a$ | SPB | 0 | $a$ | i | 6 | 7 |
| | +1 | STB | 0 | $a + 10$ | j | 14 | 15 |
| | +2 | STB | 0 | $a + 11$ | k | 18 | 19 |
| | +3 | SPB | 0 | $a + 3$ | L | 6 | 7 |
| | +4 | SPB | 0 | $a + 4$ | PL | 6 | 7 |
| | +5 | SPB | 0 | $a + 5$ | S | 6 | 7 |
| | +6 | SPB | 0 | $a + 6$ | OB    i PREG $\triangledown$ OP code | 6 | 7 |
| | +7 | SPB | 0 | $a + 7$ | $(L + i) = $ RPARENS | 6 | 7 |
| | +8 | SPB | 0 | $a + 1$ | j | 6 | 7 |
| | +9 | SPB | 0 | $a + 2$ | k | 6 | 7 |
| | +10 | ADD | $a + 8$ | one | $j + 1$ | 10 | 11 |
| | +11 | ADD | $a + 9$ | one | $k + 1$ | 14 | 15 |
| | +12 | LO7 | $a + 5$ | $a + 9$ | s + k address | 10 | 11 |
| | +13 | THS, M | 0 | $a + 12$ | $(s + k)$ - PREC $\triangledown$ OP code | 14 | 15 |
| | +14 | LO1 | $a + 13$ | OP MSK | | 18 | 19 |
| | +15 | LO6 | $a + 14$ | LPARENS MSK | | 22 | 23 |
| | +16 | EQZ | $a + 15$ | $a + 17$ | LPARENS YES | 26 | - |
| | +17 | CONS | | HALO | | - | 1 |
| | +18 | NEQ | $a + 15$ | $a + 19$ | | 26 | - |
| | +19 | CONS | | IPSWICH | | - | 1 |
| | +20 | CONS | | one | | - | 1 |
| | +21 | CONS | | OP MSK | | - | 1 |
| | +22 | CONS | | LPARENS MSK | | - | 1 |
| GEORGE | $a$ | SPB | 0 | $a$ | i | 6 | 7 |
| | +1 | ADD | $a + 15$ | one | $j = j + 1$ | 10 | 11 |
| | +2 | ADD | $a + 14$ | one | $k = k + 1$ | 10 | 11 |
| | +3 | SPB | 0 | $a + 3$ | L | 6 | 7 |
| | +4 | SPB | 0 | $a + 4$ | PL | 6 | 7 |
| | +5 | SPB | 0 | $a + 5$ | S | 6 | 7 |
| | +6 | SPB | 0 | $a + 6$ | | 6 | 7 |
| | +8 | SPB | 0 | $a + 7$ | $(L + i) = $ OPERATOR | 6 | 7 |
| | +7 | LO1 | $a + 8$ | PREC MSK | OPERATOR PREC | 10 | 11 |
| | +9 | THS | $a + 14$ | $a + 5$ | $(s + k)$ | 10 | 11 |
| | +10 | LO1 | $a + 9$ | PREC MSK | $(s + k)$ operator PREC TOP | 14 | 15 |
| | +11 | SUB | $a + 7$ | $a + 10$ | OP PREC - (S + k) OP PREC | 17 | 18 |

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | | Remarks | Time | |
|---|---|---|---|---|---|---|---|
| | | | | | | P | R |
| GEORGE (cont) | +12 | GTE | $\alpha$ + 11 | $\alpha$ + 13 | | 21 | - |
| | +13 | CONS | | JOE | | - | 1 |
| | +14 | SPB | 0 | $\alpha$ + 2 | k | 6 | 7 |
| | +15 | SPB | 0 | $\alpha$ + 1 | j | 6 | 7 |
| | +16 | LO7 | $\alpha$ + 4 | $\alpha$ + 15 | PL + j | 10 | 11 |
| | +17 | STO | $\alpha$ + 9 | $\alpha$ + 16 | (s + k) → PL + J | 14 | 15 |
| | +18 | LTZ | $\alpha$ + 11 | $\alpha$ + 23 | | 21 | - |
| | +19 | BRG | $\alpha$ + 18 | $\alpha$ + 18 | | 54 | 55 |
| | +20 | NEQ | $\alpha$ + 19 | $\alpha$ + 21 | | 58 | - |
| | +21 | CONS | | GEORGE | | - | 1 |
| | +22 | CONS | | PREC MSK | | - | 1 |
| | +23 | CONS | | SP | | | |
| | +24 | CONS | | ONE | | | |
| HALO | $\alpha$ | STB | 0 | $\alpha$ + 10 | i | 14 | 15 |
| | +1 | STB | 0 | $\alpha$ + 8 | j | 14 | 15 |
| | +2 | SPB | 0 | $\alpha$ + 2 | k | 6 | 7 |
| | +3 | SPB | 0 | $\alpha$ + 3 | L | 6 | 7 |
| | +4 | SPB | 0 | $\alpha$ + 4 | PL | 6 | 7 |
| | +5 | SPB | 0 | $\alpha$ + 5 | S | 6 | 7 |
| | +6 | SPB | 0 | $\alpha$ + 6 | OB | 6 | 7 |
| | +7 | SPB | 0 | $\alpha$ + 1 | j = j - 1 to counter j = j + 1 | 6 | 7 |
| | +8 | SUB | $\alpha$ + 7 | one | | 10 | 11 |
| | +9 | SPB | 0 | $\alpha$ | i = i + 1 | 6 | 7 |
| | +10 | ADD | $\alpha$ + 9 | one | | 10 | 11 |
| | +11 | MWT | | ABLE | | 18 | - |
| | +12 | CONS | | ONE | | | |
| IPSWICH | $\alpha$ | SPB | 0 | $\alpha$ | i | 6 | 7 |
| | +1 | SPB | 0 | $\alpha$ + 10 | j | 6 | 7 |
| | +2 | SPB | 0 | $\alpha$ + 11 | k | 6 | 7 |
| | +3 | SPB | 0 | $\alpha$ + 3 | L | 6 | 7 |
| | +4 | SPB | 0 | $\alpha$ + 4 | PL | 6 | 7 |
| | +5 | SPB | 0 | $\alpha$ + 5 | S | 6 | 7 |
| | +6 | SPB | 0 | $\alpha$ + 6 | OB | 6 | 7 |
| | +7 | SPB | 0 | $\alpha$ + 7 | (L + i) = RPARENS | 6 | 7 |
| | +8 | SPB | 0 | $\alpha$ + 13 | (s + k) - PREC 9 OP case | 6 | 7 |
| | +9 | LO7 | $\alpha$ + 4 | $\alpha$ + 1 | PL + j | 10 | 11 |
| | +10 | STO | $\alpha$ + 9 | $\alpha$ + 9 | (s + k) → PL + j | 14 | 15 |
| | +11 | NEQ | $\alpha$ + 9 | $\alpha$ + 15 | | 14 | 15 |
| | +12 | BRG | $\alpha$ + 11 | $\alpha$ + 18 | Bring SP $\alpha$ + 18 | 58 | 59 |
| | +13 | NEQ | $\alpha$ + 12 | $\alpha$ + 14 | | 61 | - |
| | +14 | CONS | | FOX | | - | 1 |
| | +15 | CONS | | SP | | | |

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | | Remarks | Time P | Time R |
|------|---|------|------|------|---------|---|---|
| JOE | $\alpha$ | ADD | $\alpha + 9$ | one | $i = i + 1$ | 10 | 11 |
| | +1 | SUB | $\alpha + 8$ | one | $j = j - 1$ | 10 | 11 |
| | +2 | SUB | $\alpha + 10$ | one | $k = k - 1$ | 10 | 11 |
| | +3 | SPB | 0 | $\alpha + 3$ | L | 6 | 7 |
| | +4 | SPB | 0 | $\alpha + 4$ | PL | 6 | 7 |
| | +5 | SPB | 0 | $\alpha + 5$ | S | 6 | 7 |
| | +6 | SPB | 0 | $\alpha + 6$ | OB | 6 | 7 |
| | +7 | SPB | 0 | $\alpha + 8$ | $(L + i)$ = OPERATOR | 6 | 7 |
| | +8 | SPB | 0 | $\alpha + 1$ | j | 6 | 7 |
| | +9 | SPB | 0 | $\alpha$ | i | 6 | 7 |
| | +10 | SPB | 0 | $\alpha + 2$ | k | 6 | 7 |
| | +11 | LO7 | $\alpha + 5$ | $\alpha + 2$ | $s + k - 1$ | 14 | 15 |
| | +12 | STO | $\alpha + 7$ | $\alpha + 11$ | $(L + c) \rightarrow s + k - 1$ | 18 | 19 |
| | +13 | MWT | | ABLE | | 22 | - |
| | +14 | CONS | | one | | - | 1 |
| KOALA | $\alpha$ | SPB | 0 | $\alpha$ | i | 6 | 7 |
| | +1 | SPB | 0 | $\alpha + 1$ | j | 6 | 7 |
| | +2 | SPB | 0 | $\alpha + 2$ | k | 6 | 7 |
| | +3 | SPB | 0 | $\alpha + 2$ | L | 6 | 7 |
| | +4 | SPB | 0 | $\alpha + 4$ | PL | 6 | 7 |
| | +5 | SPB | 0 | $\alpha + 5$ | S | 6 | 7 |
| | +6 | SPB | 0 | $\alpha + 6$ | OB | 6 | 7 |
| | +7 | SPB | 0 | $\alpha + 7$ | $(L + i)$ = OPERATOR = RTERM | 6 | 7 |
| | +8 | SPB | 0 | $\alpha + 10$ | $(S + k) \neq$ NULL | 6 | 7 |
| | +9 | LO6 | $\alpha + 4$ | $\alpha + 1$ | $PL + j$ | 10 | 11 |
| | +10 | STO | $\alpha + 8$ | $\alpha + 9$ | $(S + k) \rightarrow PL + j$ | 14 | 15 |
| | +11 | NEQ | $\alpha + 9$ | $\alpha + 15$ | | 14 | 15 |
| | +12 | BRG | $\alpha + 11$ | $\alpha + 18$ | | 58 | 59 |
| | +13 | NEQ | $\alpha + 12$ | $\alpha + 14$ | | 61 | - |
| | +14 | CONS | | DOG | | - | 1 |
| | +15 | CONS | | SP | | | |
| SP | $\alpha$ | SPB | 0 | $\alpha$ | i | 6 | 7 |
| | +1 | SPB | 0 | $\alpha + 1$ | j | 6 | 7 |
| | +2 | SPB | 0 | $\alpha + 2$ | k | 6 | 7 |
| | +3 | SPB | 0 | $\alpha + 3$ | L | 6 | 7 |
| | +4 | SPB | 0 | $\alpha + 4$ | PL | 6 | 7 |
| | +5 | SPB | 0 | $\alpha + 5$ | S | 6 | 7 |
| | +6 | SPB | 0 | $\alpha + 6$ | OB | 6 | 7 |
| | +7 | SPB | 0 | $\alpha + 8$ | $(s + k)$ i PREC $\nabla$ OP Code | 6 | 7 |
| | +8 | LOI | $\alpha + 7$ | OP MSK | | 10 | 11 |
| | +9 | LO6 | $\alpha + 8$ | ABS MSK | | 14 | 15 |
| | +10 | LO4 | $\alpha + 8$ | NEG MSK | | 14 | 15 |

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | Remarks | Time P | R |
|---|---|---|---|---|---|---|
| SP (cont) | +11 | MPY | $\alpha+9$ | $\alpha+10$ | | 18 | 19 |
| | +12 | EQZ | $\alpha+11$ | $\alpha+13$ | | 22 | - |
| | +13 | CONS | | SPU | | - | 1 |
| | +14 | NEQ | $\alpha+11$ | $\alpha+15$ | | 22 | - |
| | +15 | CONS | | SPB | | - | 1 |
| | +16 | BRG | $\alpha+14$ | $\alpha+14$ | SPB $\alpha+14$ | 21 | 22 |
| | +17 | BRG | $\alpha+12$ | $\alpha+11$ | SPB $\alpha+11$ | 21 | 22 |
| | +18 | LO7 | $\alpha+16$ | $\alpha+17$ | | 25 | 26 |
| | | CONS | | OP MSK | | - | 1 |
| | | CONS | | ABS MSK | | - | 1 |
| | | CONS | | NEG MSK | | - | 1 |
| SPB | $\alpha$ | SPB | 0 | $\alpha$ | i | 6 | 7 |
| | +1 | SPB | 0 | $\alpha+1$ | j | 6 | 7 |
| | +2 | SPB | 0 | $\alpha+2$ | k | 6 | 7 |
| | +3 | SPB | C | $\alpha+3$ | L | 6 | 7 |
| | +4 | SPB | 0 | $\alpha+4$ | PL | 6 | 7 |
| | +5 | SPB | 0 | $\alpha+5$ | S | 6 | 7 |
| | +6 | SPB | 0 | $\alpha+6$ | OB | 6 | 7 |
| | +7 | LO1 | $\alpha+11$ | imsk | i - - - | 14 | 15 |
| | +8 | STB | 216 | $\alpha+7$ | imsk | 18 | 19 |
| | +9 | LO1 | $\alpha+12$ | imsk | | 26 | 27 |
| | +10 | STB | 216 | $\alpha+9$ | | 30 | 31 |
| | +11 | THS, M | $\alpha+1$ | $\alpha+4$ | (PL + j) | 10 | 11 |
| | +12 | THS, M | $\alpha+8$ | $\alpha+4$ | N(PL + j) | 22 | 23 |
| | +13 | THS, M | $\alpha+10$ | $\alpha+4$ | N(N(PL + j)) | 33 | 34 |
| | +14 | THS | $\alpha+11$ | $\alpha+15$ | TAB   OP CODE   GENO Address | 14 | 15 |
| | +15 | CONS | TAB | TAB | | - | 1 |
| | +16 | MWT | $\alpha+14$ | $\alpha+14$ | START A GEN PROG | 18 | - |
| | +17 | MPY | $\alpha$ | ● | i● | 10 | 11 |
| | +18 | LO7 | $\alpha+6$ | $\alpha+17$ | OB + i● | 14 | 15 |
| | +19 | LO7 | $\alpha+18$ | VBIT | Ri | 18 | 19 |
| | +20 | LO7 | $\alpha+4$ | $\alpha+1$ | PL + j | 10 | 11 |
| | +21 | STO | $\alpha+19$ | $\alpha+20$ | Ri → PL + j | 22 | 23 |
| | +22 | CONS | | ● | | - | 1 |
| | +23 | CONS | | VBIT | | - | 1 |
| GENO1 | $\alpha$ | SPB | 0 | $\alpha+11$ | (PL + j) | 6 | 7 |
| | +1 | SPB | 0 | $\alpha+12$ | N(PL + j): B operand | 6 | 7 |
| | +2 | SPB | C | $\alpha+13$ | N(N(PL + j)) A operand | 6 | 7 |
| | +3 | SPB | 0 | $\alpha+18$ | OB + i● --OB i●- | 6 | 7 |
| | +4 | SPB | 208 | $\alpha+4$ | --OP Code | 6 | 7 |
| | +5 | RIM | 0 | $\alpha+10$ | F- | 18 | 19 |
| | +6 | RIM | 1 | $\alpha+10$ | DOUBLE | 18 | 19 |

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | | Remarks | Time P | Time R |
|---|---|---|---|---|---|---|---|
| GENO1 (cont) | +7 | ADD | α + 3 | one | O2 + ix + 1 | 10 | 11 |
| | +8 | ADD | α + 3 | two | + 2 | 10 | 11 |
| | +9 | ADD | α + 3 | three | + 3 | 10 | 11 |
| | +10 | THS | α + 4 | +M | | 14 | 15 |
| | +11 | CONS | | OCT | | - | 1 |
| | +12 | START | | G10 | B | 3 | - |
| | +13 | START | | G10 | A | 3 | - |
| | +14 | BRG | α + 12 | 6 | B  RDM or STB | 29 | 30 |
| | +15 | BRG | α + 13 | 6 | A  RDM or STB | 29 | 30 |
| | +16 | LO1 | α + 1 | ADDMSK | address of B operand | 10 | 11 |
| | +17 | LO1 | α + 2 | ADDMSK | address of A operand | 10 | 11 |
| | +18 | LO7 | α + 14 | α + 16 | RDM or STB B | 33 | 34 |
| | +19 | LO7 | α + 15 | α + 17 | RDM or STB A | 33 | 34 |
| | +20 | LO1 | α + 8 | i⬤MSK | i⬤+ for A | 14 | 15 |
| | +21 | LO1 | α + 9 | i⬤MSK | i⬤+ for B | 14 | 15 |
| | +22 | STB | ?2 | α + 20 | | 18 | 19 |
| | +23 | LO7 | α + 22 | α + 21 | | 22 | 23 |
| | +24 | LO7 | α + 5 | α + 23 | F - A addresses B | 26 | 27 |
| | +25 | STO | α + 24 | α + 3 | | 30 | 31 |
| | +26 | STO | α + 6 | α + 7 | | 22 | 23 |
| | +27 | STO | α + 19 | c + 8 | | 37 | 38 |
| | +28 | STO | α + 18 | α + 9 | | 37 | 38 |
| | +29 | CONS | one | | | - | 1 |
| | +30 | CONS | two | | | - | 1 |
| | +31 | CONS | three | | | - | 1 |
| | +32 | CONS | ADDMSK | | | - | 1 |
| | +33 | CONS | i ⬤ MSK | | | - | 1 |
| | +34 | CONS | EQMSK | | | - | 1 |
| G10 | α | SPR | 0 | -11 | | t | ? |
| | +1 | LO1 | α | INDEXMSK | | 10 | 11 |
| | +2 | EQZ | α + 1 | α + 5 | | 14 | - |
| | +3 | NEQ | α + 1 | α + 4 | | 14 | - |
| | +4 | CONS | | G11 | | - | 1 |
| | +5 | CONS | | G12 | | - | 1 |
| | +6 | BRG | α + 2 | 1 | | 22 | 23 |
| | +7 | CONS | | INDEX MSK | | - | 1 |
| G11 | α | RDM | | RI | RI = RDM | 16 | 17 |
| G12 | α | RDM | | SI | SI = STB | 16 | 17 |
| GENO2 | α | SPB | 0 | α + 9 | (PL + j) OPERATOR | 6 | 7 |
| | +1 | SPB | 0 | α + 10 | N(PL + j) OPERAND | 6 | 7 |
| | +2 | SPB | 0 | α + 15 | OR + i⬤ | 6 | 7 |
| | +3 | SPB | 0 | α + 11 | OP CODE | 6 | 7 |

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | | Remarks | Time P | Time R |
|---|---|---|---|---|---|---|---|
| GENO2 (cont) | +4 | RIM | 0 | α + 11 | LO6 or L10 | 14 | 15 |
| | +5 | RIM | 1 | α + 13 | LO6 or L10 | 14 | 15 |
| | +6 | RIM | 2 | α + 14 | STB 223 or CONS 0 | 14 | 15 |
| | +7 | ADD | α + 2 | one | OB + i● + 1 | 10 | 11 |
| | +8 | ADD | α + 2 | two | + 2 | 10 | 11 |
| | +9 | ADD | α + 2 | three | + 3 | 10 | 11 |
| | +10 | ADD | α + 2 | four | + 4 | 10 | 11 |
| | +11 | THS | α + 4 | α + 12 | ABS or NEG | 10 | 11 |
| | +12 | CONS | | OCT | | - | 1 |
| | +13 | START | | G20 | | 3 | - |
| | +14 | BRG | α + 13 | 6 | RDM or STB | 38 | 39 |
| | +14a | BRG | α + 13 | 7 | RDM or STB | 38 | 39 |
| | +16 | LO6 | α + 3 | ABSMSK | | 13 | 14 |
| | +17 | EQZ | α + 16 | α + 19 | | 17 | - |
| | +18 | NEQ | α + 16 | α + 20 | | 17 | - |
| | +19 | CONS | | ABS | | - | 1 |
| | +20 | CONS | | NEG | | - | 1 |
| | +21 | BRG | α + 17 | 4 | | 33 | 34 |
| | +22 | BRG | α + 17 | 5 | | 33 | 34 |
| | +23 | BRG | α + 17 | 6 | | 33 | 34 |
| | +24 | STO | α + 29 | α + 2 | | 44 | 45 |
| | +25 | STO | α + 30 | α + 7 | | 44 | 45 |
| | +26 | STO | α + 21 | α + 8 | | 37 | 38 |
| | +27 | STO | α + 14 | α + 9 | | 42 | 43 |
| | +28 | STO | α + 14a | α + 10 | | 42 | 43 |
| | +29 | LO7 | α + 4 | α + 22 | | 40 | 41 |
| | +30 | LO7 | α + 5 | α + 23 | | 40 | 41 |
| | +31 | CONS | | one | | - | 1 |
| | +32 | CONS | | two | | - | 1 |
| | +33 | CONS | | three | | - | 1 |
| | +34 | CONS | | four | | - | 1 |
| | +35 | CONS | | ABS MSK | | - | 1 |
| G20 | α | SPB | c | α + 1 | OPERAND | 6 | 7 |
| | +1 | LO1 | α | INDEXMSK | | 10 | 11 |
| | +2 | EQZ | α + 1 | α + 5 | | 14 | 15 |
| | +3 | NEQ | α + 1 | α + 4 | | 14 | 15 |
| | +4 | CONS | | G21 | | - | 1 |
| | +5 | CONS | | G22 | | - | 1 |
| | +6 | BRG | α + 2 | 1 | RDM or STB A | 24 | 25 |
| | +7 | ADD | α + 6 | one | RDM or STB A + 1 | 31 | 32 |
| | +8 | CONS | | IND MSK | | - | 1 |
| | +9 | CONS | | one | | - | 1 |

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | | Remarks | Time P | Time R |
|---|---|---|---|---|---|---|---|
| G21 | $a$ | RDM | | RI | RI RDM - | 17 | 18 |
| G22 | $a$ | RDM | | SI | SI STB - | 17 | 18 |
| SPU | $a$ | SPB | 0 | $a$ | i | 6 | 7 |
| | +1 | SPB | 0 | $a + 1$ | j | 6 | 7 |
| | +2 | SPB | 0 | $a + 2$ | k | 6 | 7 |
| | +3 | SPB | 0 | $a + 3$ | L | 6 | 7 |
| | +4 | SPB | 0 | $a + 4$ | PL | 6 | 7 |
| | +5 | SPB | 0 | $a + 5$ | S | 6 | 7 |
| | +6 | SPB | 0 | $a + 6$ | OB | 6 | 7 |
| | +7 | LO1 | $a + 9$ | imsk | | 14 | 15 |
| | +8 | STB | 216 | $a + 7$ | | 18 | 19 |
| | +9 | THS, M | $a + 1$ | $a + 4$ | $(PL + j)$ | 10 | 11 |
| | +10 | THS, M | $a + 8$ | $a + 4$ | $n(PL + j)$ | 22 | 23 |
| | +11 | THS | $a + 9$ | $a + 12$ | OP CODE GENO address | 14 | 15 |
| | +12 | CONS | | TAB | | - | 1 |
| | +13 | MWT | | $a + 11$ | Start A GENERATOR PROG | 18 | - |
| | +14 | MPY | $a$ | ● | i● | 10 | 11 |
| | +15 | LO7 | $a + 6$ | $a + 14$ | OB + i● | 14 | 15 |
| | +16 | LO7 | $a + 15$ | VBIT | | 18 | 19 |
| | +17 | LO7 | $a + 4$ | $a + 1$ | PL + j | 10 | 11 |
| | +18 | STO | $a + 16$ | $a + 17$ | Ri → PL + j | 22 | 23 |
| | +19 | CONS | | imsk | | - | 1 |
| | +20 | CONS | | VBIT | | - | 1 |
| GENO3 | $a$ | SPB | 0 | $a + 11$ | $(PL + j)$ | 6 | 7 |
| | +1 | SPB | 0 | $a + 12$ | N(PL + j) B | 6 | 7 |
| | +2 | SPB | 0 | $a + 13$ | N(N(PL + j)) A | 6 | 7 |
| | +3 | SPB | 0 | $a + 18$ | OB + i● | 6 | 7 |
| | +4 | SPB | 208 | $a + 14$ | OP Code | 6 | 7 |
| | +5 | RIM | 0 | $a + 10$ | BRG 1 | 14 | 15 |
| | +6 | RIM | 1 | $a + 10$ | BRG 2 | 14 | 15 |
| | +7 | RIM | 2 | $a + 10$ | DOUB | 14 | 15 |
| | +8 | RIM | 3 | $a + 10$ | MWT | 14 | 15 |
| | +9 | RIM | 4 | $a + 10$ | CONS | 14 | 15 |
| | +10 | THS | $a + 4$ | $a + 11$ | | 10 | 11 |
| | +11 | CONS | | OCT | | - | 1 |
| | +12 | START | $a + 13$ | | | 3 | |
| | +13 | CONS | G31 | | | - | 1 |
| | +14 | BRG | $a + 12$ | 4 | BRG ●+a 1 | 20 | 21 |
| | +15 | BRG | $a + 12$ | 5 | BRG (●+a 2 | 20 | 21 |
| | +16 | START | $a + 18$ | | | 3 | - |
| | +17 | START | $a + 18$ | | | - | 1 |
| | +18 | CONS | G10 | | | - | 1 |

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | | Remarks | Time P | R |
|------|------|------|------|------|------|------|------|
| GENO3 (cont) | +19 | BRG | α + 16 | 6 | B      RDM or STB | 33 | 34 |
| | +20 | BRG | α + 17 | 6 | A      RDM or STB | 33 | 34 |
| | +21 | LO1 | α + 1 | ADDMSK | | 10 | 11 |
| | +22 | LO1 | α + 2 | ADDMSK | | 10 | 11 |
| | +23 | LO7 | α + 19 | α + 21 | RDM 13 A add or STB B add | 37 | 38 |
| | +24 | LO7 | α + 20 | α + 22 | RDM A add or STB A add | 37 | 38 |
| | +25 | ADD | α + 3 | one | OB + 1X + 1 | 10 | 11 |
| | +26 | ADD | α + 3 | two | + 2 | 10 | 11 |
| | +27 | ADD | α + 3 | three | + 3 | 10 | 11 |
| | +28 | ADD | α + 3 | four | + 4 | 10 | 11 |
| | +29 | ADD | α + 3 | five | + 5 | 10 | 11 |
| | +30 | ADD | α + 3 | six | + 6 | 10 | 11 |
| | +31 | ADD | α + 3 | seven | + 7 | 10 | 11 |
| | +32 | STO | α + 14 | α + 3 | BRG i θ + 6  1 | 24 | 25 |
| | +33 | STO | α + 15 | α + 25 | BRG i θ + 6  2 | 24 | 25 |
| | +34 | STO | α + 24 | α + 26 | RDM or STB  A | 41 | 42 |
| | +35 | STO | α + 7 | α + 27 | DOUBLE | 18 | 19 |
| | +36 | STO | α + 23 | α + 28 | RDM or STB  B | 41 | 42 |
| | +37 | STO | α + 7 | α + 29 | DOUBLE | 18 | 19 |
| | +38 | STO | α + 8 | α + 30 | MWT | 18 | 19 |
| | +39 | STO | α + 9 | α + 31 | CONS EXP ROUT | 18 | 19 |
| | +40 | CONS | | ADD MSK | | - | 1 |
| | +41 | CONS | | one | | - | 1 |
| | +42 | CONS | | two | | - | 1 |
| | +43 | CONS | | three | | - | 1 |
| | +44 | CONS | | four | | - | 1 |
| | +45 | CONS | | five | | - | 1 |
| | +46 | CONS | | six | | - | 1 |
| | +47 | CONS | | seven | | - | 1 |
| G31 | α | SPB | 0 | α + 5 | BRG  1 | 6 | 7 |
| | +1 | SPB | 0 | α + 6 | BRG  2 | 6 | 7 |
| | +2 | SPB | 88 | α + 22 | OB + i θ + 6 | 6 | 7 |
| | +3 | STB | 16 | α + 2 | - i θ + 6 - | 3 | 4 |
| | +4 | LO7 | α | α + 3 | BRG i θ + 6  1 | 10 | 11 |
| | +5 | LO7 | α + 1 | α + 3 | BRG i θ + 6  2 | 10 | 11 |
| G30 | α | SPB | 0 | -15 | A or B operand | 6 | 7 |
| | +1 | LO1 | α | INDMSK | | 10 | 11 |
| | +2 | EQZ | α + 1 | α + 4 | | 14 | - |
| | +3 | NEQ | α + 1 | α + 5 | | 14 | - |
| | +4 | CONS | | G33 | | - | 1 |
| | +5 | CONS | | G32 | | - | 1 |
| | +6 | BRG | α - 2 | 1 | | 23 | 24 |

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | | Remarks | Time P | R |
|------|------|------|------|------|------|------|------|
| G33 | a | RDM | | RI | RI RDM - | 16 | 17 |
| G32 | a | RDM | | SI | SI STB - | 16 | 17 |
| GENO4 | a | SPB | 0 | a + 11 | (PL + j) | 6 | 7 |
| | +1 | SPB | 0 | a + 12 | N(PL + j)   B operand | 6 | 7 |
| | +2 | SPB | 0 | a + 13 | N(N(PL + j)  A operand | 6 | 7 |
| | +3 | SPB | 0 | a + 18 | OB + ix   OB ix | 6 | 7 |
| | +4 | SPB | 208 | a + 14 | OP code | 6 | 7 |
| | +5 | RIM | 0 | a + 10 | STO | 14 | 15 |
| | +6 | RIM | 1 | a + 10 | DOUBLE | 14 | 15 |
| | +7 | ADD | a + 3 | one | OB + i● + 1 | 10 | 11 |
| | +8 | ADD | a + 3 | two | + 2 | 10 | 11 |
| | +9 | ADD | a + 3 | three | + 3 | 10 | 11 |
| | +10 | THS | a + 4 | a + 11 | | 10 | 11 |
| | +11 | CONS | | OCT | | - | 1 |
| | +12 | LOI | a + 2 | ADDMSK | A address | 10 | 11 |
| | +13 | LOI | a + 1 | ADDMSK | B address | 10 | 11 |
| | +14 | STB | 72 | a + 5 | STO OB + i● + 2 OB + i● + 3 | 14 | 15 |
| | +15 | LO7 | a + 13 | a + 14 | | 18 | 19 |
| | +16 | STO | a + 14 | a + 3 | STO i● + 2 i● + 3 | 18 | 19 |
| | +17 | STO | a + 6 | a + 7 | DOUBLE | 18 | 19 |
| | +18 | STO | a + 12 | a + 8 | A address | 14 | 15 |
| | +19 | STO | a + 13 | a + 9 | B address | 14 | 15 |
| | +20 | CONS | | one | | | |
| | +21 | CONS | | two | | | |
| | +22 | CONS | | three | | | |
| | +23 | CONS | | ADDMSK | | | |
| ABS | a | SPB | 72 | a + 3 | | 6 | 7 |
| | +1 | SPB | 0 | a + 9 | - - - OB i● 3 | 6 | 7 |
| | +2 | SPB | 0 | a + 10 | - - - OB i● 4 | 6 | 7 |
| | +3 | SPB | 0 | a + 6 | STB 223 | 6 | 7 |
| | +4 | LO7 | a + 3 | a + 1 | STB 223 i● + 3 | 10 | 11 |
| | +5 | LO7 | a | a + 1 | i● + 2 i● + 3 | 10 | 11 |
| | +6 | LO7 | a | a + 2 | i● + 2 i● + 4 | 10 | 11 |
| NEG | a | SPB | 72 | a + 9 | i● + 3 | 6 | 7 |
| | +1 | SPB | 72 | a + 10 | i● + 4 | 6 | 7 |
| | +2 | SPB | 0 | a + 9 | i● + 3 | 6 | 7 |
| | +3 | SPB | 0 | a + 10 | i● + 4 | 6 | 7 |
| | +4 | SPB | 0 | a + 6 | CONS 0 | | |
| | +5 | LO7 | a | a + 2 | i● + 3 i● + 3 | 10 | 11 |
| | +6 | LO7 | a + 1 | a + 3 | i● + 4 i● + 4 | 10 | 11 |
| OCT | 1 | FAD | i● + 2 | i● + 1 | | | |
| | 2 | DOUBLE | | | | | |

## TABLE XIII-3 - COMPILER PROGRAM (Continued)

| Item | | Instruction | | | Remarks | Time | |
|---|---|---|---|---|---|---|---|
| | | | | | | P | R |
| OCT (cont) | 3 | FSU | i⚫ + 2 | i⚫ + 3 | | | |
| | 4 | DOUBLE | | | | | |
| | 5 | FDV | i⚫ + 2 | i⚫ + 3 | | | |
| | 6 | DOUBLE | | | | | |
| | 7 | FMP | i⚫ + 2 | i⚫ + 3 | | | |
| | 8 | DOUBLE | | | | | |
| | 9 | STC | i⚫ + 2 | i⚫ + 3 | EQUAL | | |
| | 10 | DOUBLE | | | | | |
| | α | LO6 | iX + 2 | iX + 3 | ABS | | |
| | +1 | LO6 | iX + 2 | iX + 4 | | | |
| | +2 | STB | 223 | + 3 | | | |
| | +3 | A address | | | | | |
| | +4 | A + 1 address | | | | | |
| | α | L1C | iX + 3 | iX + 3 | NEG | | |
| | +1 | L10 | iX + 1 | iX + 4 | | | |
| | +2 | CONS | 0 | | | | |
| | +3 | A address | | | | | |
| | +4 | A + 1 address | | | | | |
| | α | BRG | i⚫ + 6 | 1 | EXP | | |
| | +1 | BRG | i⚫ + 6 | 2 | | | |
| | +2 | RDM | A address | | | | |
| | +3 | DOUBLE | | | | | |
| | +4 | RDM | B address | | | | |
| | +5 | DOUBLE | | | | | |
| | +6 | MWT | | | | | |
| | +7 | CCNS | EXP ROUT | | | | |

## TABLE XIII-4 - OBJECT PROGRAM

| Address | Instruction | | | Remarks |
|---|---|---|---|---|
| OB + 5⊗ | FMP | 7⊗ + 2 | 7⊗ + 3 | A · B |
| +1 | Double | | | |
| +2 | RDM | A | | |
| +3 | RDM | B | | |
| OB + 9⊗ | FMP | 9⊗ + 2 | 9⊗ + 3 | C · D |
| +1 | Double | | | |
| +2 | RDM | C | | |
| +3 | RDM | D | | |
| OB : 7⊗ | FAD | 12⊗ + 2 | 12⊗ + 3 | A · B   C · D |
| +1 | Double | | | |
| +2 | STB | 0 | 7⊗ | A · B |
| +3 | STB | 0 | 9⊗ | C · D |
| OB + 15⊗ | BRG | 15⊗ + 6 | 1 | E · F |
| +1 | BRG | 15⊗ + 6 | 2 | |
| +2 | RDM | E | | |
| +3 | Double | | | |
| +4 | RDM | F | | |
| +5 | Double | | | |
| +6 | MWT | 15⊗ + 7 | | |
| +7 | CONS | Exprout | | |
| OB + 18⊗ | LO6 | 18⊗ + 2 | 18⊗ + 3 | ABS I |
| +1 | LO6 | 18⊗ + 2 | 18⊗ + 4 | |
| +2 | STB | 223 | 18⊗ + 3 | |
| +3 | RDM | I | | |
| +4 | RDM | I + 1 | | |
| OB + 17⊗ | FAD | 17⊗ + 2 | 17⊗ + 3 ' | E ↑ F + ABS I |
| +1 | Double | | | |
| +2 | STB | 0 | 15⊗ | |
| +3 | STB | 0 | 18⊗ | |
| OB + 12⊗ | FMP | 12⊗ + 2 | 12⊗ + 3 | (A · B   C · D)   (E↑F ABS I) |
| +1 | Double | | | |
| +2 | STB | 0 | 7X | |
| +3 | STB | 0 | 17X | |
| OB + 24⊗ | BRG | 24⊗ + 6 | 1 | J ↑ K |
| +1 | BRG | 24⊗ + 6 | 2 | |
| +2 | BRG | J | | |
| +3 | Double | | | |

TABLE XIII-4 - OBJECT PROGRAM (Continued)

| Address | Instruction | | | Remarks |
|---|---|---|---|---|
| +4 | RDM | K | | |
| +5 | Double | | | |
| +6 | MWT | 24⊕ + 7 | | |
| +7 | CONS | Exprout | | |
| OB + 28⊕ | FDV | 28⊕ + 2 | 28⊕ + 3 | L/M |
| +1 | Double | | | |
| +2 | RDM | L | | |
| +3 | RDM | M | | |
| OB + 26⊕ | FSU | 26⊕ + 2 | 26⊕ + 3 | J ↑ K - L/M |
| +1 | Double | | | |
| +2 | STB | 0 | 24⊕ | |
| +3 | STB | 0 | 28⊕ | |
| OB + 23⊕ | L10 | 23⊕ + 3 | 23⊕ + 3 | NEG P |
| +1 | L10 | 23⊕ + 4 | 23⊕ + 4 | |
| +2 | CONS | 0 | | |
| +3 | RDM | P | | |
| +4 | RDM | P + 1 | | |
| OB + 37⊕ | PMP | 27⊕ + 2 | 37⊕ + 3 | Q · R |
| +1 | Double | | | |
| +2 | RDM | Q | | |
| +3 | RDM | R | | |
| OB + 35⊕ | FAD | 35⊕ + 2 | 35⊕ + 3 | NEG P + Q · R |
| +1 | Double | | | |
| +2 | STB | 0 | 23⊕ | |
| +3 | STB | 0 | 37⊕ | |
| OB + 31⊕ | FDV | 31⊕ + 2 | 31⊕ + 3 | (J ↑ K - L/M)/(NEG P + Q · R) |
| +1 | Double | | | |
| +2 | STB | 0 | 26⊕ | |
| +3 | STB | 0 | 35⊕ | |
| OB + 21⊕ | FAD | 21⊕ + 2 | 21⊕ + 3 | (A · B+C · D) · (E ↑ F + ABS I) |
| +1 | Double | | | |
| +2 | STB | 0 | 12⊕ | (J ↑ K - L/M)/(NEG P + Q · R) |
| +3 | STB | 0 | 31⊕ | |
| OB + 2⊕ | STO | 2⊕ + 2 | 2⊕ + 3 | |
| +1 | Double | | | |
| +2 | CONS | 21⊕ | | |
| +3 | CONS | Z | | |

1. INTRODUCTION

The Machine I parallel processor described in Appendix VI has some disadvantages. While many tasks could be run concurrently, each task is sequential and communication between tasks is difficult.

In Machine II, each task (instruction block) can have concurrently operating instructions and communication between tasks is better. Machine I has the advantage of better machine utilization since a programmer can automatically introduce concurrency without spending time setting up new tasks. Machine II has the advantage of transferring results between tasks without memory references.

Instructions generally consist of an operation code and two operand fields. When a task is started, any instruction in the task will be performed when its operands are available; thus many instructions in a task could be executed simultaneously.

2. BRIEF DESCRIPTION OF MACHINE

Machine II consists of a multiaccess merging-separating memory (see Appendices VI and VII) connected to I/O devices and a multiprocessor control unit (MPC) as diagrammed in Figure XIV-1. The MPC stores a large set of instructions (on the order of 1000) and fetches their operands and feeds them to processors for execution. The number of processors may be in the hundreds. The channel between the MPC and the memory is large enough to permit the transfer of 1024 words at one time.

3. WORD FORMATS

In memory, a word consists of a 24-bit address and a 32-bit data field.
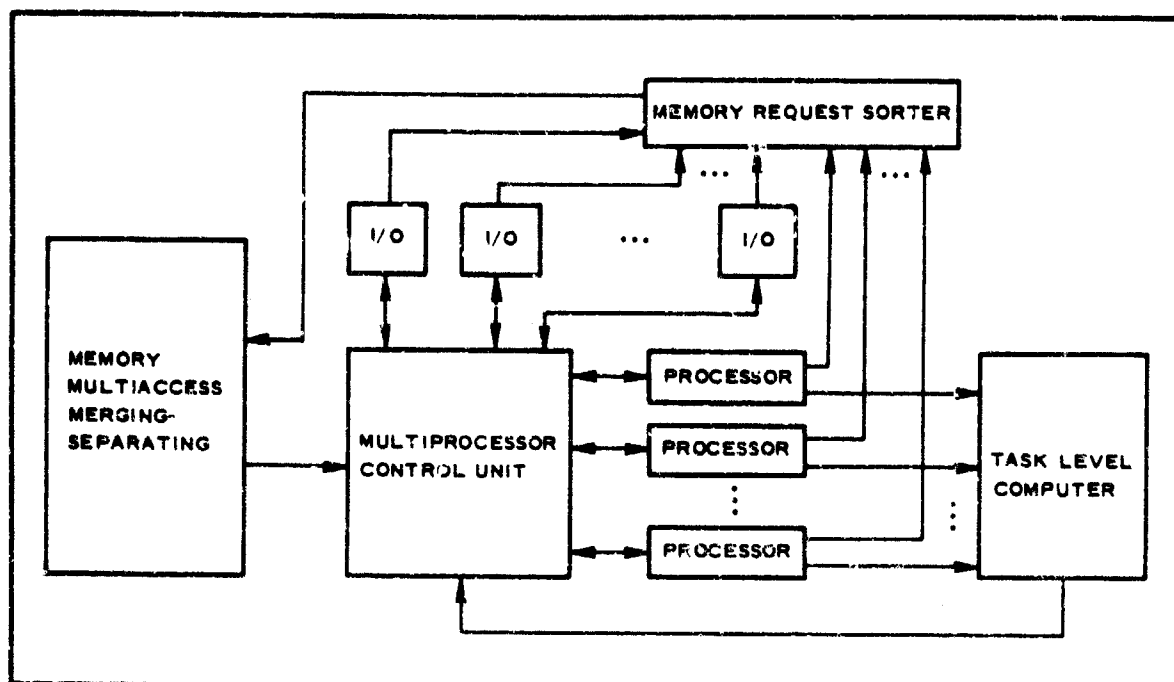
Figure XIV-1 - Block Diagram of Machine II

Programming is done with 16-bit addresses. Up to 256 different programs may be running concurrently, each with its own protected set of 65,536 addresses. A program cannot reference another's address except through the monitor. A given address may be empty, contain one word, or contain several words. When more than one word is at the same address, they are arranged in order of their contents. A normal reference to an address will provide the word whose contents is the least of all words at the address. A special threshold memory reference allows the retrieval of the least word at an address whose contents are not below a specified threshold. This allows instant retrieval of an item in a table.

An integer is in the range $1 - 2^{31}$ to $2^{31} - 1$ and is written in normal ONEs-complement form (sign bit is 0 for positive numbers, 1 for negative numbers). Several integers at the same address will be ordered with positive integers first in increasing order then negative integers in decreasing order of magnitude. Certain integers are shown below:

| Integer | Representation |
|---|---|
| $2^{31} - 1$ | 0111 1111 1111 1111 1111 1111 1111 1111 |
| $2^{31} - 2$ | 0111 1111 1111 1111 1111 1111 1111 1110 |
| 2 | 0000 0000 0000 0000 0000 0000 0000 0010 |
| 1 | 0000 0000 0000 0000 0000 0000 0000 0001 |
| +0 | 0000 0000 0000 0000 0000 0000 0000 0000 |
| -1 | 1111 1111 1111 1111 1111 1111 1111 1110 |
| -2 | 1111 1111 1111 1111 1111 1111 1111 1101 |
| $2 - 2^{31}$ | 1000 0000 0000 0000 0000 0000 0000 0001 |
| $1 - 2^{31}$ | 1000 0000 0000 0000 0000 0000 0000 0000 |

A floating-point number consists of a fraction sign, an 11-bit exponent (the fraction is multiplied by any power of 2 between $2^{-1024}$ and $2^{1023}$) and a 20-bit fraction. A double-length floating-point number has a 52-bit fraction. A positive floating-point number has a fraction sign of 0 and the exponent is biased; for example, 0 represents $2^{-1024}$. The fraction is in the range -1 to 1. A negative floating-point number is formed by complementing every bit (sign, exponent, and fraction). Certain single-length floating-point numbers are shown on the next page. Note that with this representation a table of positive normalized floating-point numbers can be put in order simply by putting their representations in order; this format allows threshold searches on positive normalized floating-point numbers. Negative normalized floating-point numbers will be put in descending order.

Instructions are read into the MPC in blocks of from 1 to 256 instructions apiece. An instruction block is stored at one address. The instruction format is:

| Number (8 bits) | Operation code (8 bits) | A (8 bits) | B (8 bits) |
|---|---|---|---|

| Number | Representation (single-length) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $0.5 \times 2^{1023}$ | 0111 | 1111 | 1111 | 1000 | 0000 | 0000 | 0000 | 0000 |
| $0.5 \times 2^{1}$ | 0100 | 0000 | 0001 | 1000 | 0000 | 0000 | 0000 | 0000 |
| $0.5 \times 2^{0}$ | 0100 | 0000 | 0000 | 1000 | 0000 | 0000 | 0000 | 0000 |
| $0.5 \times 2^{-1}$ | 0011 | 1111 | 1111 | 1000 | 0000 | 0000 | 0000 | 0000 |
| $0.5 \times 2^{-1024}$ | 0000 | 0000 | 0000 | 1000 | 0000 | 0000 | 0000 | 0000 |
| 0 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| $-0.5 \times 2^{-1024}$ | 1111 | 1111 | 1111 | 0111 | 1111 | 1111 | 1111 | 1111 |
| $-0.5 \times 2^{-1}$ | 1100 | 0000 | 0000 | 0111 | 1111 | 1111 | 1111 | 1111 |
| $-0.5 \times 2^{0}$ | 1011 | 1111 | 1111 | 0111 | 1111 | 1111 | 1111 | 1111 |
| $-0.5 \times 2^{1}$ | 1011 | 1111 | 1110 | 0111 | 1111 | 1111 | 1111 | 1111 |
| $-0.5 \times 2^{1023}$ | 1000 | 0000 | 0000 | 0111 | 1111 | 1111 | 1111 | 1111 |

The number identifies the instruction in the block; each instruction in a block is given a unique number from 0 to 255. The operation code identifies the operation while A and B usually identify operands.

An instruction block is read into the MPC in one piece and the instructions in it may be executed in any order (any instruction is executed whenever the requisite number of operands are available). The operands for instructions may be memory words, results of other instructions in the same block, results of instructions in the block that caused a block to be read into the MPC, or results of instructions in any block that a block causes to be read into the MPC. Several instruction blocks may be in the MPC at one time.

4. OPERATIONS

a. General

Each operation specifies one or two operands and has options that allow

erasure of its operands. When the result of an operation is being used by several other operations, one and only one of the other operations should erase it, as discussed under Item 4, i below.

b. Arithmetic and Logic

(1) Operands

The operands in any of these operations are the results of the operations numbered a and b in the same block as the operation.

(2) Fixed Point

$$\text{ADD} \quad \text{Result is (a)} + \text{(b)}$$

$$\text{SUB} \quad \text{Result is (a)} - \text{(b)}$$

$$\text{MPY} \quad \text{Result is (a)} * \text{(b)}$$

$$\text{DVD} \quad \text{Result is (a)} / \text{(b)}$$

$$\text{MOD} \quad \text{Result is (a) Mod (b)}$$

Arithmetic is done modulo $2^{32} - 1$. Negative zeros are never generated.

The operation ADD, A is similar to ADD except that the result in (a) is erased. This also is true for ADD, B and ADD, AB and all other fixed-point operations. There are a total of 20 fixed-point operations. Overflows will be flagged.

(c) Floating Point

$$\text{FAD} \quad \text{(a)} + \text{(b)}$$

$$\text{FSU} \quad \text{(a)} - \text{(b)}$$

$$\text{FMP} \quad \text{(a)} * \text{(b)}$$

$$\text{FDV} \quad \text{(a)} / \text{(b)}$$

Operands and results are normal single-length floating-point numbers. Each operation has three erase options indicated, for example, by FAD, A; FAD, B; FAD, AB, which cause erasure of (a), of (b) or of (a) and (b).

respectively. There are a total of 16 floating-point operations. Overflows will be flagged.

**(4) Double-Length Floating-Point**

Any of the four floating-point operations may be made double-length by putting the operation DOUBLE in the instruction following the single-length operation; if DOUBLE is numbered n + 1, the single-length operation is numbered n. The continuation of the double-length result is numbered the same as the double operation. As an example, the program

          100   FAD, A   3   6

          101   DOUBLE   0   0

would treat the results numbered 3 and 4 as the A operand, the results numbered 6 and 7 as the B operand, and would number the resultant sum as 100 and 101. The erase A option causes erasure of the results numbered 3 and 4. There is one double operation.

**(5) Logic**

The logic operations combine the corresponding bits of A and B with any of the 16 possible Boolean functions of two variables. They are listed at the top of the next page.

Each of these operations has erase options; for example, L00, A or L00, B or L00, AB, which cause erasure of A, of B, or of A and B, respectively. There are 64 logic operations.

**(6) Conversions**

FXFL (Fixed-to-Floating) - The fixed point result numbered a is converted to floating point. Field b is unused. If this operation precedes a double operation, a double-length floating point number is created. FXFL, A erases the result numbered a.

FLFX (Floating-to-Fixed) - The floating-point result numbered a is

| | | | | |
|---|---|---|---|---|
| L00 | | 0 | | (zeros) |
| L01 | (A) | $\wedge$ | (B) | (and) |
| L02 | (A) | $\wedge$ | $(\bar{\bar{B}})$ | |
| L03 | | (A) | | |
| L04 | $(\bar{A})$ | $\wedge$ | (B) | |
| L05 | | (B) | | |
| L06 | (A) | $\oplus$ | (B) | (exclusive or) |
| L07 | (A) | $\vee$ | (B) | (or) |
| L10 | $(\bar{A})$ | $\wedge$ | $(\bar{B})$ | |
| L11 | $(\bar{A})$ | $\oplus$ | (B) | |
| L12 | | $(\bar{B})$ | | |
| L13 | (A) | $\vee$ | $(\bar{B})$ | |
| L14 | | $(\bar{A})$ | | |
| L15 | $(\bar{A})$ | $\vee$ | (B) | |
| L16 | $(\bar{A})$ | $\vee$ | $(\bar{B})$ | |
| L17 | | 1 | | (ones) |

converted to fixed-point. Overflow is flagged. FLFX, A erases the operand a.

DLFX (Double-Length Floating-to-Fixed) - The double-length result numbered a and a + 1 is converted to fixed-point. Overflow is flagged. DLFX, A erases operands a and a + 1.

(7) Conclusions

In each of these operations, the operands are results of other instructions in the same block and the result is left numbered the same as the instruction that caused it.

## c. Shift

In the shift operations, the a field contains a shift constant in the range
0 to 255. A shift constant in the range 0 to 63 means a left-end-around
shift of from 0 to 63 places. A shift constant in the range 64 to 127
means a left end-off shift of from 0 to 63 places (zeros are written into
the right side of the result). A shift constant in the range 128 to 191
means a left end-off shift of 0 to 63 places (ones are written into the right
side of the result). A shift constant in the range 192 to 255 means a right
end-off shift (the sign bit is written into the left side of the result).

In summary:

$$0 \leq a \leq 63 \qquad \text{Shift left end-around a places,}$$
$$64 \leq a \leq 127 \qquad \text{Shift left end-off a-64 places (write zeros)}$$
$$128 \leq a \leq 191 \qquad \text{Shift left end-off a-128 places (write ones)}$$
$$192 \leq a \leq 255 \qquad \text{Shift right end-off a-192 places (write sign)}$$

The operand is specified in the b field. It is always left alone or erased
and the result of the shift instruction is the shifted operand. For the STB
operation, the operand is in the same block as the instruction (Shift This
Block). For the SPB operation, the operand is numbered b in the previous
block (Shift Previous Block). (The previous block is that block that con-
tained the start instruction that started this block.) For the SPR oper-
ation, the operand is in the previous block b places relative to the start
instruction that started this block (Shift Previous Relative) (b is added to
the number of the start instruction modulo 256). The STB, B; SPB, B;
and SPR, B operations cause erasure of the operand.

Any shift may be made a double-length shift by following the instruction
with a DOUBLE operation. As an example, the program

$$43 \qquad \text{SPR, B} \qquad 70 \qquad 13$$

$$44 \qquad \text{DOUBLE} \qquad 0 \qquad 0$$

would take the results of the instructions in the previous block, which are

the 13th and 14th instructions following the start instruction, erase them, put them together in a double-length word (the 13th result to the left), shift them left end-off 6 places (6 places on the left are lost, 6 places in the right half travel to the left half and 6 zeros are written in the right half), then number the resultant halves as 43 and 44 in this block.

In all left shift operations, the result is flagged if overflow occurs.

d. Bring

The shift operations SPB and SPR allow a block to retrieve an item from the block that started it. The BRING operation (BRG) allows a block to retrieve an item from a block it starts. The a field identifies the block by specifying the start instruction in this block which started the block containing the operand. The b field specifies the number of the result in that block. The result of the BRG is the particular operand. BRG, B erases the operand in the started block.

Field a of a BRG or BRG, B operation may refer to a conditional start instruction that was not satisfied. In this case, the first start instruction or satisfied conditional start instruction whose number is above a is selected and reference made to the block it started. If no such start exists, an interrupt occurs.

e. Memory References

(1) RDM (Read Memory)

This has as a result the word in memory whose address is (a, b) (a and b are read as one 16-bit field to form the memory address). If there is more than one word at the address, the one whose contents is least is chosen. RDM, M erases the word in memory.

(2) RIM (Read Indirect Memory)

The 16 right-most bits of the result b in this block is used as the memory address after being incremented by an amount from -128 to 127. The increment is in the a field of this instruction. RIM, B erases the base

-365-

memory address, RIM, M erases the word in memory, and RIM, BM erases both.

### (3)   THS (Threshold Search)

This has as a result the word in memory in a given address whose contents are just above a threshold.   The memory address is specified by the right-most 16 bits of the result in this block numbered b.   The threshold is the result numbered a.

THS, M erases the memory word, THS, A erases the result a in this block, and THS, B erases the result b in this block.   THS, AB; THS, AM; THS, ABM; and THS, BM erase the indicated combinations of these items.

In THS, lack of the desired memory word causes a search in the higher memory addresses with the retrieval of the memory word whose contents is the least in the first nonempty memory address.   This item will be erased if the memory erase option is specified.   Care should be exercised to prevent unwanted erasures.   The result of a THS is flagged if it comes from an address different from the specified address.

### (4)   STO (Store)

This causes the storing of a result into memory.   The memory address is the right-most 16 bits of the result numbered b.   The result to be stored is that numbered a.   STO, A erases the a-result, STO, B erases the b-result, and STO, AB erases both.   The store operations themselves have no results.   No memory words are over written; any words at the specified memory address will be kept and the new memory word added to that address.

### f.   Instruction Block Starts

### (1)   START

This causes an instruction block to be put in the MPC for execution.   The memory address of the block is specified by the right-most 16 bits of the

result numbered b in this block.  START B is similar to START except that the result numbered b is erased.

(2)  Conditional Starts

The condition of a result in a block can be used to start another block with the same priority,  The a field specifies the result to be tested.  The memory address of the block to be started is contained in the right-most 16 bits of the result numbered b.  The conditional starts are:

$$GTZ \quad start \quad if \quad (a) > 0$$
$$LTZ \quad start \quad if \quad (a) < 0$$
$$GTE \quad start \quad if \quad (a) \geqq 0$$
$$LTE \quad start \quad if \quad (a) \leqq 0$$
$$EQZ \quad start \quad if \quad (a) = 0$$
$$NEQ \quad start \quad if \quad (a) \neq 0$$
$$FLG \quad start \quad if \quad (a) \quad is \quad flagged$$
$$UNF \quad start \quad if \quad (a) \quad is \quad unflagged$$

Each conditional start has erase options; for example, GTZ, A erases result a, GTZ, B erases result b, and GTZ, AB erases both.  These erasures occur whether the condition is satisfied or not.

(3)  SUPER (Supervisor)

This operation brings a supervisory routine into the MPC.  The a and b fields specify which routine.  The routine may expect to obtain certain parameters from results stored relative to the SUPER instruction.  Supervisory routines are allowed certain privileged operations denied normal routines.

(4)  M WAIT

This is an unconditional start instruction except that it is not executed until all memory operations (RDM, RIM, THS, STO) in this block have been executed.

### g.   EPB (Erase Previous Block

This causes all results in the previous block between and including a and
b to be erased.  EPR (Erase Previous Block Relative) is similar except
a and b are relative to the start instruction that started this block.

### h.   CONST Constant

The operation CONST has as a result the number stored in the a and b
fields.  Sixteen zeros are inserted in the left-most places.

### i.   Erasures

If a result is used by only one operation, that operation should erase it.
If used more than once, then it should be erased by (1) the highest num-
bered operation in the lowest priority block started by the highest num-
bered start instruction that uses it; or if no started block uses it,  then
(2) the highest numbered operation in the same block that uses it; or if no
operation in the block uses it, then (3) the highest numbered operation in
the previous block that uses it.

## 5.   EXAMPLE PROGRAMS

### a.   POLY

POLY evaluates a fourth-degree polynomial in X using double-length
floating-point arithmetic.  The coefficients $c_0$, $c_1$, $c_2$, $c_3$, $c_4$ are stored
in POLY + 1 through POLY + 10.  The variable X is obtained from the
calling block   The entry should leave the left-half of X two results ahead
of the START POLY instruction and the right-half of X one result ahead.
POLY erases X and leaves $C_0 + c_1 X + c_2 X^2 + c_3 X^3 + c_4 X^4$ in results 1
and 2.

In Figure XIV-2, Instructions 3 through 14 are executed first to obtain
the coefficients and X.  Instructions 3 and 4 erase X in the calling pro-
gram.  Instructions 15 through 20 are executed next.  On the third step,
Instructions 21 through 28 are executed.  On the fourth step, Instructions

| ADDRESS | NO. | OP | A | B | RESULT |
|---|---|---|---|---|---|
| POLY | 1 | FAD, AB | 29 | 31 | $C_0 + C_1 x + C_2 x^2 + C_3 x^3 + C_4 x^4$ |
| | 2 | DOUBLE | 0 | 0 | |
| | 3 | SPR, B | 0 | -2 | $x$ |
| | 4 | SPR, B | 0 | -1 | |
| | 5 | RDM | POLY + | 1 | $C_0$ |
| | 6 | RDM | POLY + | 2 | |
| | 7 | RDM | POLY + | 3 | $C_1$ |
| | 8 | RDM | POLY + | 4 | |
| | 9 | RDM | POLY + | 5 | $C_2$ |
| | 10 | RDM | POLY + | 5 | |
| | 11 | RDM | POLY + | 7 | $C_3$ |
| | 12 | RDM | POLY + | 8 | |
| | 13 | RDM | POLY + | 9 | $C_4$ |
| | 14 | RDM | POLY + | 10 | |
| | 15 | FMP | 3 | 3 | $x^2$ |
| | 16 | DOUBLE | 0 | 0 | |
| | 17 | FMP, A | 7 | 3 | $C_1 x$ |
| | 18 | DOUBLE | 0 | 0 | |
| | 19 | FMP, A | 13 | 3 | $C_4 x$ |
| | 20 | DOUBLE | 0 | 0 | |
| | 21 | FAD, AB | 5 | 17 | $C_0 + C_1 x$ |
| | 22 | DOUBLE | 0 | 0 | |
| | 23 | FMP, A | 0 | 15 | $C_2 x^2$ |
| | 24 | DOUBLE | 0 | 0 | |
| | 25 | FMP, AB | 3 | 15 | $x^3$ |
| | 26 | DOUBLE | 0 | 0 | |
| | 27 | FAD, AB | 11 | 19 | $C_3 + C_4 x$ |
| | 28 | DOUBLE | 0 | 0 | |
| | 29 | FAD, AB | 21 | 23 | $C_0 + C_1 x + C_3 x^2$ |
| | 30 | DOUBLE | 0 | 0 | |
| | 31 | FMP, AB | 25 | 27 | $C_3 x^3 + C_4 x^4$ |
| | 32 | DOUBLE | 0 | 0 | |
| POLY + 1 | | | | | |
| POLY + 2 | | $C_0$ | | | |
| POLY + 3 | | | | | |
| POLY + 4 | | $C_1$ | | | |
| POLY + 5 | | | | | |
| POLY + 6 | | $C_2$ | | | |
| POLY + 7 | | | | | |
| POLY + 8 | | $C_3$ | | | |
| POLY + 9 | | | | | |
| POLY + 10 | | $C_4$ | | | |

Figure XIV-2 - Example of POLY Program

| TREE | 1 | SPR, B | 0 | - 2 | N |
|---|---|---|---|---|---|
| | 2 | SPR, B | 0 | - 1 | PROG |
| | 3 | CONST | 0 | 1 | 1 |
| | 4 | START, B | 0 | 5 | START TREE + 1 WITH INDEX 1 |
| | 5 | CONST | TREE + 1 | | |

| TREE + 1 | 1 | SPR | 0 | - 1 | I |
|---|---|---|---|---|---|
| | 2 | START | -1 | 4 | START PROG WITH INDEX I |
| | 3 | SPR, B | 0 | - 3 | N |
| | 4 | SPN, B | 0 | - 2 | PROG |
| | 5 | SPR | 0 | - 1 | I |
| | 6 | LTZ, B | 13 | 9 | START TREE + 2 OF 2I < N |
| | 7 | EQZ, B | 13 | 10 | START TREE + 3 IF 2I = N |
| | 8 | GTZ, AB | 13 | 11 | START TREE + 4 IF 2I > N |
| | 9 | CONST | TREE + 2 | | |
| | 10 | CONST | TREE + 3 | | |
| | 11 | CONST | TREE + 4 | | |
| | 12 | SPR, B | 1 | - 1 | 2I |
| | 13 | SUB, A | 12 | 3 | 2I - N |

| TREE + 2 | 1 | SPR | 0 | - 3 | N |
|---|---|---|---|---|---|
| | 2 | SPR | 0 | - 2 | PROG |
| | 3 | SPR | 1 | - 1 | 2I |
| | 4 | START | 0 | 9 | START TREE + 1 WITH INDEX 2I |
| | 5 | SPR, B | 0 | - 3 | N |
| | 6 | SPR, B | 0 | - 2 | PROG |
| | 7 | SPR, B | 129 | - 1 | 2I + 1 |
| | 8 | START, B | 0 | 9 | START TREE + 1 WITH INDEX 2I + 1 |
| | 9 | CONST | TREE + 1 | | |

| TREE + 3 | 1 | SPR, B | 0 | - 4 | N |
|---|---|---|---|---|---|
| | 2 | SPR, B | 0 | - 3 | PROG |
| | 3 | SPR, B | 1 | - 2 | 2I |
| | 4 | START, B | 0 | 5 | START TREE + 1 WITH INDEX 2I |
| | 5 | CONST | TREE + 1 | | |

| TREE + 4 | 1 | EPR | -5 | - 3 | |
|---|---|---|---|---|---|

Figure XIV-3 - Example of TREE Program

29 through 32 are executed. On the fifth step, Instructions 1 and 2 are executed.

b. TREE

TREE (see Figure XIV-3) is a program that will cause N executions of a program specified as PROG, each execution with a different index I, I = 1, 2, . . . , N. PROG should be written to expect I one location ahead of its start instruction and it should erase I.

TREE is entered with

$$\alpha - 2: \quad N$$

$$\alpha - 1: \quad PROG$$

$$\alpha: \quad START \quad TREE$$

and it erases N and PROG.

TREE consists of five instruction blocks, TREE, TREE + 1, . . . , TREE + 4, and 33 instructions.

6. CONCLUSIONS

The normal operations of a multiprocessor design have been described. There will also be other operations for use by the monitor. This machine has the advantage of having a machine language wherein parallel operations can be expressed and executed easily and communication between concurrently operating portions of the programs can be accomplished.

7. OPERATIONS THAT LEAVE A RESULT

a. Fixed Point

| ADD | ADD, A | ADD, B | ADD, AB |
| SUB | SUB, A | SUB, B | SUB, AB |
| MPY | MPY, A | MPY, B | MPY, AB |
| DVD | DVD, A | DVD, B | DVD, AB |

|       |         |          |          |
|-------|---------|----------|----------|
| MOD   | MOD, A  | MOD, B   | MOD, AB  |
| FLFX  | FLFX, A | DLFX     | DLFX, A  |

b.   Floating Point (Can Be Double Length)

|       |         |          |          |
|-------|---------|----------|----------|
| FAD   | FAD, A  | FAD, B   | FAD, AB  |
| FSU   | FSU, A  | FSU, B   | FSU, AB  |
| FMP   | FMP, A  | FMP, B   | FMP, AB  |
| FDV   | FDV, A  | FDV, B   | FDV, AB  |
| FXFL  | FXFL, A |          |          |

c.   Logic

|       |         |          |          |
|-------|---------|----------|----------|
| L00   | L00, A  | L00, B   | L00, AB  |
| L01   | L01, A  | L01, B   | L01, AB  |
| L02   | L02, A  | L02, B   | L02, AB  |
| L03   | L03, A  | L03, B   | L03, AB  |
| L04   | L04, A  | L04, B   | L04, AB  |
| L05   | L05, A  | L05, B   | L05, AB  |
| L06   | L06, A  | L06, B   | L06, AB  |
| L07   | L07, A  | L07, B   | L07, AB  |
| L10   | L10, A  | L10, B   | L10, AB  |
| L11   | L11, A  | L11, B   | L11, AB  |
| L12   | L12, A  | L12, B   | L12, AB  |
| L13   | L13, A  | L13, B   | L13, AB  |
| L14   | L14, A  | L14, B   | L14, AB  |
| L15   | L15, A  | L15, B   | L15, AB  |
| L16   | L16, A  | L16, B   | L16, AB  |
| L17   | L17, A  | L17, B   | L17, AB  |

d.   Shift (Can Be Double Length)

|       |         |
|-------|---------|
| STB   | STB, B  |
| SPB   | SPB, B  |
| SPR   | SPR, B  |

-372-

e.  Bring

　　　　　　　　　　　　　BRG　　BRG, B

f.  Special

　　　　　　　　　　　　　　DOUBLE

　　　　　　　　　　　　　　CONST

g.  Memory

| RDM | RDM, M | | |
|---|---|---|---|
| RIM | RIM, M | RIM, B | RIM, BM |
| THS | THS, A | THS, B | THS, AB |
| THS, M | THS, AM | THS, BM | THS, ABM |

8.  OPERATIONS THAT LEAVE NO RESULT

a.  Erases

　　　　　　　　　　　　EPB　　EPR

b.  Store

| STO | STO, A | STO, B | STO, AB |
|---|---|---|---|

c.  Starts

| START | SUPER | M WAIT | START, B |
|---|---|---|---|
| GTZ | GTZ, A | GTZ, B | GTZ, AB |
| LTZ | LTZ, A | LTZ, B | LTZ, AB |
| GTE | GTE, A | GTE, B | GTE, AB |
| LTE | LTE, A | LTE, B | LTE, AB |
| EQZ | EQZ, A | EQZ, B | EQZ, AB |
| NEQ | NEQ, A | NEQ, B | NEQ, AB |
| FLG | FLG, A | FLG, B | FLG, AB |
| UNF | UNF, A | UNF, B | UNF, AB |

with many (on the order of 1000) parallel channels to the multiprocessor control. It is needed as a store capable of reading and writing many items of data simultaneously so that the machine is not memory-limited.

The I/O devices consist of backup memories (core storage, disk storage, drum storage, tape storage, etc.) and normal I/O units (card equipment, printers, consoles, channels to other machines, etc.). The number of I/O channels can be in the hundreds and all channels may be operating at the same time to give the machine a high I/O data rate. The faster I/O units can be connected to more than one I/O channel so more than one word could be transferred in any one cycle.

Each processor is a simple three-register arithmetic unit capable of performing the arithmetic-logic operations in the instruction set. Double-length operations are performed by connecting two adjacent processors. The result of each operation is transferred back to the MPC immediately after execution, freeing the processor for another instruction that may come from a different program; this rule simplifies the implementation of interrupt, multiprogramming, recovery from processor failure, and other matters in the machine. The number of processors may be in the hundreds.

The task level computer is used to implement a dynamic task priority scheme wherein each task can be assigned a certain percentage of machine capacity and is given execution time at regular intervals.

The memory request sorter (MRS) receives read and write requests from the I/O devices and processors, orders them by address and data fields, and transfers them to the memory.

The multiprocessor control (MPC) is the heart of the machine. In one sense, the MPC acts as a switchboard, connecting all the various parts of the machine together and allowing hundreds of data transfers to take place simultaneously. In another sense, it acts as a flexible buffer matching the data rates in all the data transfers. In still another sense, it acts

as an extensive "instruction look-ahead" unit arranging for the retrieval of instruction blocks and operands, matching the operands to their instructions, dispatching the instruction-operand sets to processors, and storing intermediate results. The MPC is a sorting memory with certain added features.

3.  MEMORY

The multiaccess parallel merging-separating memory is essentially that described in Appendix VII with a few modifications. The format of a word in memory is shown in Figure XV-2.

The X and Y fields designate six different kinds of items. The memory cycle has eight steps:

1.  Input new words, read requests, and read and erase requests

2.  Merge

3.  Flag requests and associated memory words

4.  Separate flagged items

5.  Present requested words for output

6.  Merge

7.  Flag requests and memory words to be erased

8.  Separate flagged items and erase them

| ADDRESS (24) | X (1) | DATA (32) | Y (4) |
|---|---|---|---|
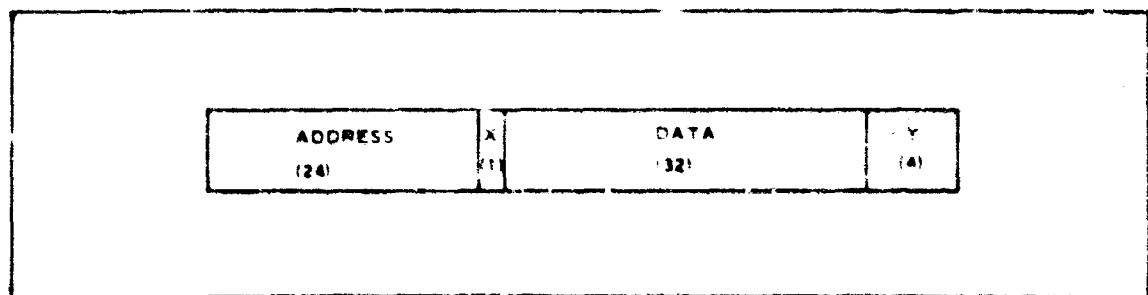
Figure XV-2 - Memory Word Format

The memory cycle is longer than described in Appendix VII because of the need for reading blocks of data (such as instruction blocks). With blocks of data, there is no convenient method for combining the reading and the erasing functions; therefore, in Step 5 the data are read but not erased and at the end of Step 8 the data are erased by overwriting it with new re- quests in Step 1 of the next cycle. An approximate cycle time can be ob- tained by assuming 150 nsec per clock period except during separates, where 250 nsec should be assumed. Using these assumptions and the fact that the even-numbered steps take n clock pulses (for a $2^n$-word memory), a cycle time of $0.8n + 0.6$ μsec is obtained; for example, a 32,768-word memory has a cycle time of about 12.6 μsec. In each cycle, 1000 items or so may be retrieved.

There are six types of items in memory designated with the following X and Y fields:

| X | Y |  |
|---|---|---|
| 0 | 0000 | Multiple read request (lower limit) |
|  | 0001 | Multiple read and erase request (lower limit) |
| 0 | 0010 | Read request |
| 0 | 0011 | Read and erase request |
| 0 | 0100 | Normal memory word |
| 1 | 0000 | Upper limit |

Each request has a corresponding upper-limit.

At the start of a cycle, the MRS presents to the lower part of memory an inverse-ordered set of requests, upper limits, and new memory words (Step 1). During Step 2, these are merged with existing memory words. During Step 3, the following items are flagged by setting their leftmost Y-field bits:

    1.   Each request and upper limit

2. Each memory word intervening between a multiple request and an upper limit

3. Each memory word that is directly above a request

During Step 4, the flagged items are separated from the unflagged and sent to the lower part of memory. During Step 5, the MPC reads the flagged items. If the number of flagged items exceeds the channel capacity, the procedure is different: the upermost upper limit in the channel is picked as a dividing point and it and all words higher than it have their flags reset while all other words are read by the MPC (the words whose flags were reset will remain in memory for the next memory cycle). During Step 6, all items are merged. During Step 7, the flags of any unerased memory words are reset. During Step 8, all flagged items are separated from the unflagged, sent to the lower part of memory, and changed to all-zero memory words (0100 in the Y field) or overwritten with new requests from the MRS.

4. PROCESSORS

Each processor is a three-register arithmetic unit and an instruction register. The instruction register contains the operation code and task identification. In each cycle, the MPC-processor interface can transfer the following.

1. Operand A from MPC to processor

2. Operand B from MPC to processor

3. Operation code and program identification (packed into one word) from MPC to processor

4. Result of last operation from processor to MPC

During each cycle, the processor performs the indicated operation. Double-length operations use two adjacent processors. One receives the upper halves of the operations and a specially flagged code (to indicate upper half), the other receives the lower halves and a specially

flagged operation code (to indicate lower ha  A connection between adjacent processors is used for the necessary interchange of data between the processors. The task identification is fed to the task level computer (see 5. below). A memory request is transferred to the memory request sorter (see 6. below).

5. TASK LEVEL COMPUTER

When a computer system is being time-shared by several tasks, a means is needed to transfer control between the tasks. The means could be hardware or software or a combination of the two. In a system with more than one processor, the implementation is complicated by the fact that a given task may be using a dynamic number of processors; to keep the processors busy, the means for processor assignment should be fast.

In Machine II, all instructions ready for execution are kept in a list in the MPC ordered by "task levels." The task level is a number assigned to an instruction block upon entry into the MPC; it governs the priority of the block relative to all other blocks in the MPC. Blocks with lower task levels are preferred to those with higher task levels. On each execution cycle, all processors interrogate the instruction list; this keeps the processors busy regardless of the changes in any one task.

The task level computer receives task identifications from the processors and uses this information to keep track of machine usage and to update task levels. The updated task levels effect the read-in priorities of new instruction blocks. The example below illustrates the scheme.

Let Machine II have four tasks, A, B, C, and D, and suppose it is desired to give Task A 50 percent of the machine capacity, Task B 30 percent, Task C 10 percent, and Task D 10 percent. Give each task an integer, $\Delta$, that is inversely related to its desired capacity. A suitable set of $\Delta$'s in this example is $\Delta_A = 3$, $\Delta_B = 5$, $\Delta_C = 15$, $\Delta_D = 15$. The task identification contains $\Delta$ as a subfield.

On each execution cycle, the task level computer increments each task level by the product of the $\Delta$ for the task and the number of processors used by the task. This information is contained in the task identifications fed from the processors. This operation causes the task level for a task to increase at a rate proportional to its current machine usage and its $\Delta$. The task levels govern the priority of the tasks in future competitions; this has the effect of keeping the task levels together since the tasks with lower levels will win future competitions, causing their levels to increase up to the higher task levels.

The example illustrates this. Let Machine II have 150 processors and assume that all tasks want to use 100 processors if given the chance. Table XV-1 shows the task levels at successive execution cycles assuming a given initial condition. Here, Machine II is used as follows:

> Task A, 1050 processor executions (50 percent)
> Task B, 650 processor executions (31 percent)
> Task C, 200 processor executions (9.5 percent)
> Task D, 200 processor executions (9.5 percent)

These percentages are close to the desired percentages (50, 30, 10, and 10). Because Task B obtained slightly more capacity than desired, its task level is higher so that in future competitions it loses out. In the long run, the actual machine usage approaches the desired machine usage.

All processors were kept busy each execution cycle (there were always enough instructions for it to do), the machine usage approximated the desired machine usage, and every task obtained access to the processors once in awhile. Thus, this seems like a good assignment procedure.

As time progresses, the task levels increase; to prevent overflow, a constant is subtracted from all task levels whenever the highest task level overflows. The easiest constant to pick is the power of two represented by the highest bit in the task level field.

## TABLE XV-1 - TASK LEVELS AT SUCCESSIVE EXECUTION CYCLES
## ASSUMING A GIVEN INITIAL CONDITION

| Task A ($\Delta = 3$) | | Task B ($\Delta = 5$) | | Task C ($\Delta = 15$) | | Task D ($\Delta = 15$) | |
|---|---|---|---|---|---|---|---|
| Level | Processors | Level | Processors | Level | Processors | Level | Processors |
| 100 | 100 | 200 | 50 | 300 | | 400 | |
| 400 | 50 | 450 | | 300 | 100 | 400 | |
| 550 | | 450 | 50 | 1800 | | 400 | 100 |
| 550 | 100 | 700 | 50 | 1800 | | 1900 | |
| 850 | 100 | 950 | 50 | 1800 | | 1900 | |
| 1150 | 100 | 1200 | 50 | 1800 | | 1900 | |
| 1450 | 100 | 1450 | 50 | 1800 | | 1900 | |
| 1750 | 50 | 1700 | 100 | 1800 | | 1900 | |
| 1900 | 50 | 2200 | | 1800 | 100 | 1900 | |
| 2050 | 50 | 2200 | | 3300 | | 1900 | 100 |
| 2200 | 100 | 2200 | 50 | 3300 | | 3400 | |
| 2500 | 50 | 2450 | 100 | 3300 | | 3400 | |
| 2650 | 100 | 2950 | 50 | 3300 | | 3400 | |
| 2950 | 100 | 3200 | 50 | 3300 | | 3400 | |
| 3250 | | 3450 | | 3300 | | 3400 | |

The task level computer consists of a small sorting memory and a set of serial adders. It sorts the task identifications arriving from the processors and whenever two words for the same task are sorted together, an adder adds the two fields and erases one of the words. Over several cycles, the necessary additions to each task level word are made. The task level words are periodically fed to the MPC. Note that the task level word is not updated instantaneously but will usually lag behind; the effect of this is to introduce some "overshoot" in the process but this will not have any effect over the long run

6. MEMORY REQUEST SORTER

Because the memory is a merging-separating memory, the memory requests must be presented to it in ordered fashion. The memory request sorter (MRS) gathers all memory requests from the processors and I/O units and orders them. The ordered set is presented to the memory during Step 1 of its cycle.

Each write request is one word with the format shown in Figure XV-3. When this is put in the memory, it will act as a normal memory word. Each read request consists of two words, an upper limit and a lower limit. The lower limit format shown in Figure XV-3 is where Y is the code for the particular type of request (see Item 3) and the threshold is all zeros except for a threshold search operation. The upper limit format shown in Figure XV-3 is where the MPC information indicates where the data retrieved by the request should go in the MPC.
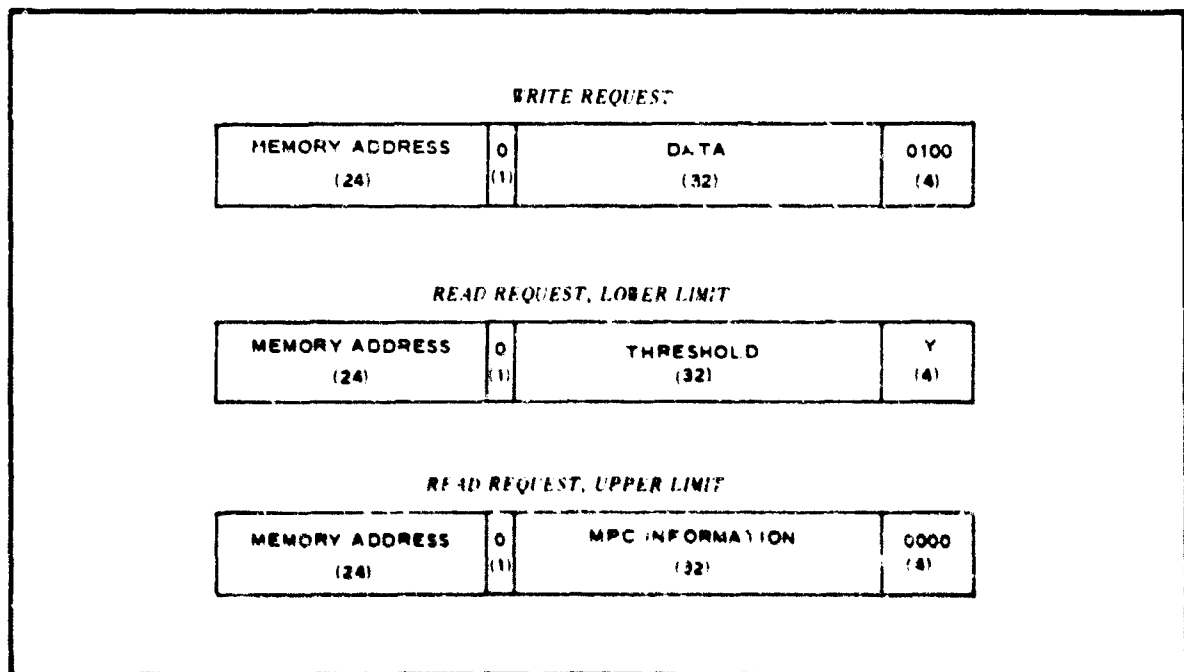


Figure XV-3 - Memory Request Formats

## 7. MULTIPROCESSOR CONTROL

### a. General

The multiprocessor control (MPC) consists of a sorting memory with logic between adjacent words to cause certain changes in the words. There are three kinds of interfaces with the MPC: I/O devices, processors, and memory channels. The uppermost end of the MPC is the I/O region with each I/O device connected to one word in the region. Immediately below the I/O region is the processor region with each processor connected to three consecutive words in the region. The lowermost end of the MPC is the memory region with each memory channel connected to three consecutive MPC words.

The MPC cycle consists of a sort phase during which all MPC words are sorted, and a transfer phase during which the interfaces read and/or write into their corresponding words and certain words are interpreted and modifications made.

The following five kinds of words are in the MPC: $\alpha$, $\beta$, $\gamma$, $\delta$, and $\epsilon$, with the formats as shown in Figure XV-4. During the transfer phase, these words are interpreted as follows.

$\alpha$ word: If the word above an $\alpha$ word is an $\epsilon$ word with the same A field, then move the F field of the $\alpha$ word into the A and B fields of the same word and copy the C and H fields of the $\epsilon$ word into the C, F, and G fields. The $\epsilon$ word is undisturbed while the $\alpha$ word is changed to an $\epsilon$ word with new A, B, and H fields. If the word above is not an $\epsilon$ word or does not have the same A field, the $\alpha$ word is undisturbed.

$\beta$ word: Same as the $\alpha$ word except that the $\epsilon$ word is erased (C = 100 and all other fields cleared to zeros).

$\gamma$ word: If the two words above a $\gamma$ word are $\epsilon$ words with the same A fields, set the left-most bit of the $\gamma$ word and the two $\epsilon$ words to 1; otherwise, leave all words alone.
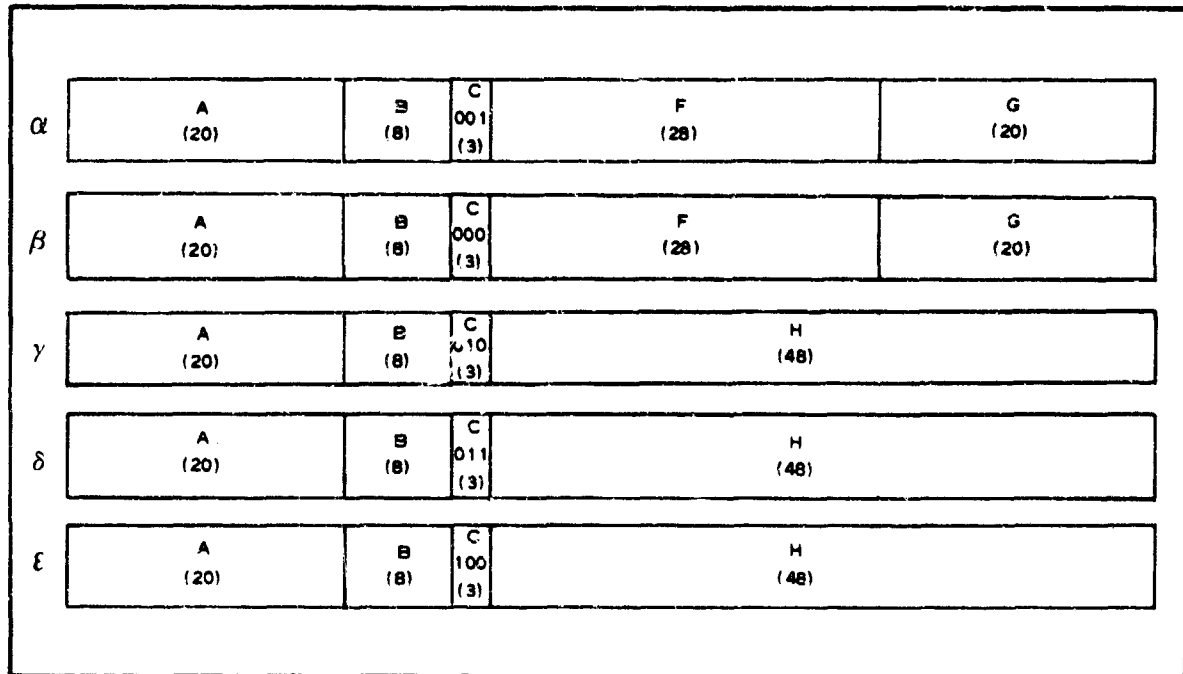
Figure XV-4 - Multiprocessor Control Word Formats

δ word: If the word above a δ word is an ε word with the same A field, its left-most bit is set to 1 and the δ word is erased (C = 100 and all other fields cleared to zeros); otherwise, set the left-most bit of the δ word to 1.

ε word: Not interpreted except in relation to adjacent α, β, γ, or δ words.

The set of words in the MPC is divided into seven regions. The size of these regions varies with time and one or more of them may be empty at a particular time. The left most three bits of each word indicates the region it is in. The regions are listed below with the three-bit codes.

> 111 - I/O region
> 110 - Processor region
> 101 - Result region
> 011 - I/O buffer region

010 - Instruction region
001 - Pointer region
000 - Memory region

As described above, the MPC interfaces are connected to the I/O region, processor region. and memory region. The size of the I/O region is fixed.

The operation of the MPC can be described by showing the actions that occur for words retrieved from memory (words from memory may be words for output or instructions or data), words from I/O devices, and words from processors.

b.   Output Words

An output device requests a block of consecutive words from memory by putting a read request or read and erase request in the MRS. The upper limit of the request contains the output device code. When the block appears on the memory interface to the MPC, an £ word is written for each word. The upper and lower limits become erased words while every word in between has the output device code preceded by the I/O buffer region code (011) written in the A field. The 24-bit memory address and the 32-bit data field of this word are put in the B and H fields. These MPC words travel to the I/O buffer region in the next sort phase.

The I/O buffer region is ordered by I/O device number, memory address, and data field.

c.   Channel Words

Every I/O channel (whether the I/O device is operating or not) inserts into its corresponding MPC word in the I/O region a fixed δ word with the I/O buffer region code (011) and the I/O device code in the A field. This word travels to the I/O buffer region and either sends back the least word in the device buffer, if there is one, or sends back itself if there isn't one. In his way, each output device reads its own buffer.

I/O units are started by putting specially flagged control words in their buffers.

d. Instructions

The upper limit of an instruction-block read request contains the MPC block assignment for the block, the program ID, and the MPC address of the start instruction that caused it to be read in. When an instruction block arrives over the memory - MPC interface, each word causes three MPC words to be formed. Two are operand requests and one is the OP code - program ID word (dummy words are formed in place of operand requests for instructions that have less than two operands). The operand request format is in Figure XV-5.

X is either a zero or a one depending on whether the operand should be erased or not. The operand request is an $\alpha$ or a $\beta$ word so that when the desired operand appears in the result region it is copied and the operand request sent to the instruction region. The OP code-program ID word is a $\gamma$ word so that when the two operand requests return to it, all three are sent to the processor region.

When a new instruction block is read in, a pointer word containing the MPC block address and address of the start instruction is put in the pointer region. This is used by SPB and SPR operations to find operands.

| 101 (3) | OPERAND ADDRESS (17) | 00000000 (8) | 00X (3) | 010 (3) | INSTRUCTION ADDRESS (25) | 0 ——————— 0 (30) |
|---|---|---|---|---|---|---|
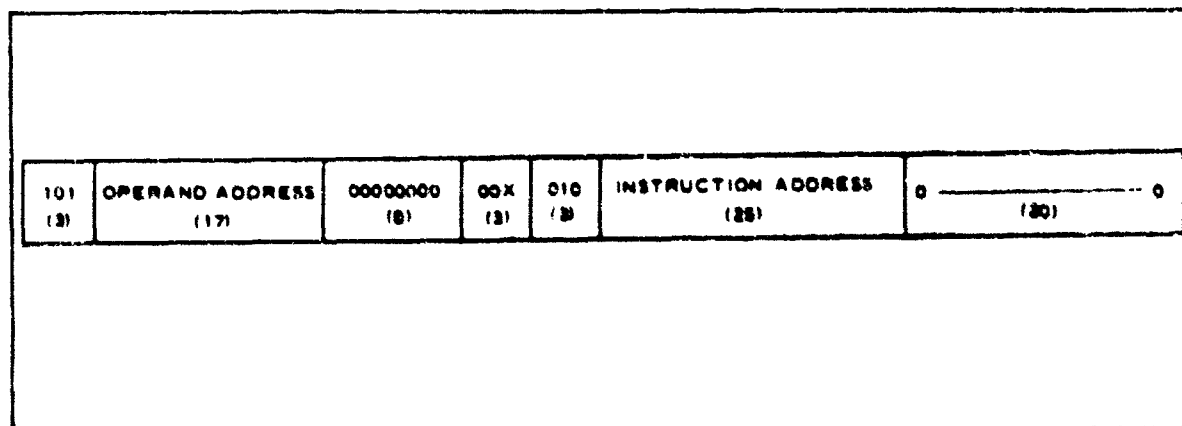
Figure XV-5 - Operand Request Format

e. Data

Data requests are sent to the result region. The upper limit contains the MPC address.

f. Processor Results

Results of instructions are sent to the result region addressed appropriately.

g. Summary

This describes the MPC. Generally speaking, $\epsilon$ words contain data while $\alpha$, $\beta$, $\gamma$, and $\delta$ words act as data requests. The I/O region is fixed in length by guaranteeing a fixed number of words with the I/O region code (if a $\delta$ word finds nothing to send to the I/O region, it sends itself).

An MPC of 8192 words requires 91 steps ($1/2 \times 13 \times 14 = 91$) in its sort phase and 1 step in its transfer phase. At 150 nsec per step, the MPC cycle is 13.8 usec. A good assumption to time out example problems then is 13.8 usec per MPC cycle. Figure XV-6 shows the timing charts.

8. CONCLUSIONS

The various parts of Machine II have been described. The main difference between it and Machine I is the multiprocessor control (MPC), which allows automatic dynamic processor assignments, the ability to code parallel programs without specifically assigning new processors, and the ability to crosstalk between parallel programs. This enhances the efficiency of the machine in many programs.

ARITHMETIC AND LOGIC (4 CYCLES MINIMUM)

```
          ≥ 0
"A" IN    "A" IN
RESULT    INSTRUCTION         "A", "B", AND    RESULT IN    RESULT IN
REGION    REGION              INSTRUCTION      PROCESSOR    RESULT
                          ≥ 0 IN PROCESSOR     REGION       REGION
"B" IN    "B" IN              REGION
RESULT    INSTRUCTION
REGION    REGION
```
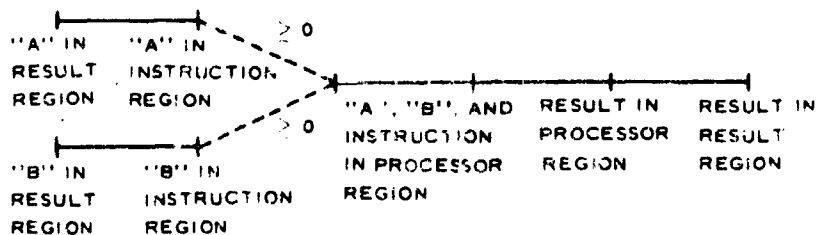
CONVERSIONS AND STB (4 CYCLES)

```
"B" IN    "B" IN      "B" AND          RESULT IN   RESULT IN
RESULT    INSTRUC-    INSTRUCTION      PROCESSOR   RESULT
REGION    TION        IN PROCESSOR     REGION      REGION
          REGION      REGION
```
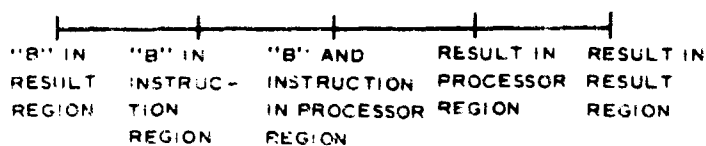
SPB, SPR, BRG (7 CYCLES MINIMUM)

```
                ≥ 0                              ≥ 0
INSTRUCTION     POINTER    POINTER TO    "B" IN    "B" IN      "B" AND          RESULT IN   RESULT IN
READ-IN         IN         INSTRUCTION   RESULT    INSTRUC-    INSTRUCTION      PROCESSOR   RESULT
                POINTER    REGION        REGION    TION        IN PROCESSOR     REGION      REGION
                REGION                             REGION      REGION
```

RDM (3 CYCLES)

```
INSTRUC-    INSTRUC-     DATA IN     DATA IN
TION        TION IN      MEMORY      RESULT
READ-IN     PROCESSOR    REGION      REGION
            REGION
```

RIM 4 RIM (4 CYCLES)

```
"B" IN    "B" IN      "B" AND      DATA IN    DATA IN
RESULT    INSTRUC-    INSTRUC-     MEMORY     RESULT
REGION    TION        TION IN      REGION     REGION
          REGION      PROCESSOR
                      REGION
```

THS (4 CYCLES MINI

```
"A" IN    "A"
RESULT    INSTI
REGION    REGI

"B" IN    "B"
RESULT    INSTR
REGION    REGI
```

STO (4 CYCLES MINI

```
"A" IN    "A"
RESULT    INST
REGION    REG

"B" IN    "B"
RESULT    INST
REGION    REG
```

START (3 CYCLES)

```
"B" IN    "B"
RESULT    INST
REGION    TIO
          REG
```

CONDITION TO START

```
"A" IN    "A"
RESULT    INSTR
REGION    REGI

"B" IN    "B"
RESULT    INST
REGION    REG
```

NOTE

TIME IS EXPRESSED IN MPC CYCLES
ABOUT 13.8 μSEC.

DOTTED PATHS ARE "WAITS" FOR VARIOUS
CONDITIONS AND MAY BE ZERO LENGTH

A

THS (4 CYCLES MINIMUM)

SUPER (2 CYCLES)

"A" IN RESULT REGION — "A" IN INSTRUCTION REGION ≥ 0

"B" IN RESULT REGION — "B" IN INSTRUCTION REGION ≥ 0

"A", "B", AND INSTRUCTION IN PROCESSOR REGION

DATA IN MEMORY REGION

DATA IN RESULT REGION

INSTRUC- TION READ-IN

INSTRUC- TION IN PROCESSOR REGION

SUPERVISORY ROUTINE IN MEMORY REGION, POINTER IN POINTER REGION
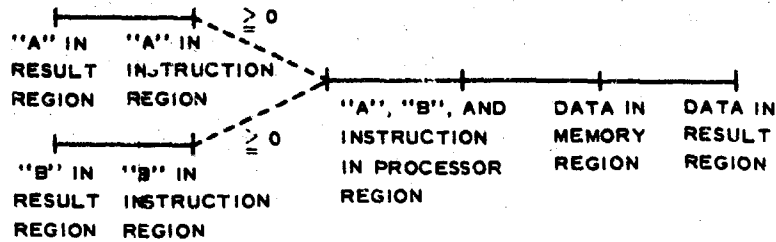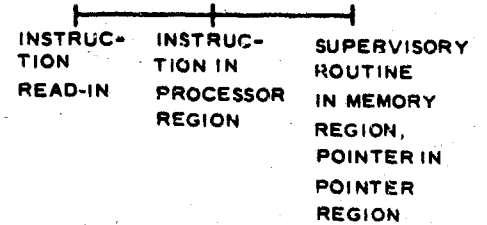
STO (4 CYCLES MINIMUM TO BE REREAD)

"A" IN RESULT REGION — "A" IN INSTRUCTION REGION ≥ 0

"B" IN RESULT REGION — "B" IN INSTRUCTION REGION ≥ 0

"A", "B", AND INSTRUCTION IN PROCESSOR REGION

DATA IN MEMORY

DATA IN MEMORY REGION

DATA IN RESULT REGION ≥ 0

WHEN ITEM IS REREAD INTO THE MPC

RESULT IN PROCESSOR REGION   RESULT IN RESULT REGION

START (3 CYCLES)

"B" IN RESULT REGION

"B" IN INSTRUC- TION REGION

"B" AND INSTRUC- TION IN PROCESSOR REGION

INSTRUCTION BLOCK READ-IN TO MEMORY REGION AND POINTER IN PROCESSOR REGION

EPB, EPR (3 CYCLES)

INSTRUC- TION READ-IN

INSTRUC- TION IN PROCESSOR REGION

ERASE REQUESTS IN PROCESSOR REGION

ERASE REQUESTS TO RESULT REGION

CONDITIONAL STARTS (3 CYCLES MINIMUM)

"A" IN RESULT REGION — "A" IN INSTRUCTION REGION ≥ 0

"B" IN RESULT REGION — "B" IN INSTRUCTION REGION ≥ 0

"A", "B", AND INSTRUCTION IN PROCESSOR REGION

INSTRUCTION BLOCK READ-IN TO MEMORY REGION AND POINTER IN POINTER REGION (IF CONDITION SATISFIED)

CONST (1 CYCLE)

INSTRUCTION READ-IN

CONSTANT IN RESULT REGION

RESSED IN MPC CYCLES μSEC).

THS ARE "WAITS" FOR VARIOUS AND MAY BE ZERO LENGTH.

Figure XV-6 - Timing Charts

B

# APPENDIX XVI - PARALLEL NONNUMERIC PROCESSING

## 1. INTRODUCTION

Nonnumeric processing is discussed in general along with the characteristics that are present in present-day machines and those characteristics that are desirable in a parallel nonnumeric processor. Ways of implementing these characteristics by means of sorting memories are discussed. The detailed design of a parallel nonnumeric processor awaits further study.

## 2. NONNUMERIC PROCESSING

The words "numeric" and "nonnumeric" when applied to data processing problems are misnomers. A look at typical numeric and nonnumeric problems reveals the distinguishing characteristic - the addressing of data. In a typical numeric problem, most items of data are addressed by their unique labels (addresses); this can be called "explicit addressing." In a typical nonnumeric problem, most items of data are addressed by their properties; this can be called "implicit addressing." This can be seen when a typical numeric programming language, such as FORTRAN, in which each item is referred to by a unique label, is compared with a typical nonnumeric programming language, such as for list processing, in which a typical operation might be the searching of a list structure for a set of items meeting a given pattern.

## 3. CLASSES OF PROPERTIES

In general, the properties by which data are implicitly addressed fall into three classes:

1. Any property dependent on an item of data per se; for example, the property of being greater than or less than a threshold or the property of having certain of its bits

matching a pattern. This class usually is called content-addressing.

2. Any maximum or minimum property such as the property of being the largest or smallest item in a set. This is referred to here as limit-addressing.

3. Any property dependent on "neighborhoods." When these occur in a nonnumeric problem, there is a structure (topology) imposed on a set of data such as lists, trees, matrices, list structures, etc. A typical property by which items may be addressed might be the satisfying of a subpattern. This is referred to here as structure-addressing.

Properties from more than one of these classes may be used in a single search. For instance, one of the search patterns mentioned in Appendix X is a string of five items (structure-addressing), the first, third, and fifth of which are operators and the second and fourth of which are variables (content-addressing) and in which the precedence of the third item is greater than that of the first item and not less than that of the fifth item (limit-addressing). These properties are separated into these classes because the implementations of searches for properties usually differ.

## 4. SOME PRESENT-DAY NONNUMERIC PROCESSORS

Most conventional computers are capable only of explicit addressing of data. A few (the CDC-1604, for example) can perform equality search or threshold search operations by which a contiguous table in memory can be content-addressed; these operations search sequentially and thus are practical only for small tables. To make the solution of some nonnumeric problems more amenable on a conventional computer, a number of languages are available of which LISP, IPL-V, and SNOBOL are examples. In essence, these languages arrange the storage of data more efficiently so that structure-addressing is easier; link fields in items represent the neighborhoods. The amount of time spent in housekeeping in these programs lowers their potential to small nonnumeric problems.

Content-addressing memories (CAMs) can perform content-addressing very well since all of memory is interrogated at the same time. By adding a fast facility to indicate the presence or nonpresence of responses, limit-addressing (maximum and minimum searches) also is performed very well. Structure addressing can be added with multiple comparands (see Item 5, below). Single-comparand CAM's might require long times to do certain structure-addressing problems.

If the problems to be solved are limited to those with a certain topology, the response store of a CAM could be interconnected in that topology and a machine obtained that would solve problems in that class very well. Two machines with this organization are the Illiac III at the University of Illinois[1, a] and the SOLOMON.[2] Both of these have the topology of a square array. On problems that fit the square array, these machines do very well while on other problems they lose much of their speed. There are many different topologies present in nonnumeric problems; for example, lists, list structures, trees, arrays, and graphs. In many problems, the topology changes as computation proceeds, hence a machine with a fixed topology will be limited in purpose. The topology of any practical nonnumeric problem can be represented by a graph with weighted directed links; nodes represent items, and links represent the connections or relations between neighboring items (the link weight shows the kind of relation). As is shown under Item 5 below, content-addressing can be changed to structure-addressing so an organization based on graphs will have great utility.

5. CONTENT-ADDRESSING BY STRUCTURE-ADDRESSING

Given a processor capable of representing any topology, one can implement content-addressing. The technique is to separate each item into its separate fields and connect the fields by weighted links to show where they

---

[a] Superior numbers in the text refer to items in the List of References under Item 11, Page 405.

occur, and then coalesce any equal-valued items. Each item contains only one field and its value is unique so its value can be used as a label or address by which it can be explicitly addressed. The example that follows exhibits this technique. Suppose there are the following eight 3-field items to be content-addressed:

| | | |
|---|---|---|
| A | 1 | 3 |
| B | 2 | 1 |
| C | 1 | 2 |
| D | 3 | 3 |
| E | 2 | 2 |
| F | 1 | 1 |
| G | 3 | 1 |
| H | 2 | 1 |

The three fields of each item are separated and connected by a link of Weight 2 between the first and second fields and a link of Weight 3 between the first and third fields. Then all equal-valued items are coalesced. The resulting graph is:
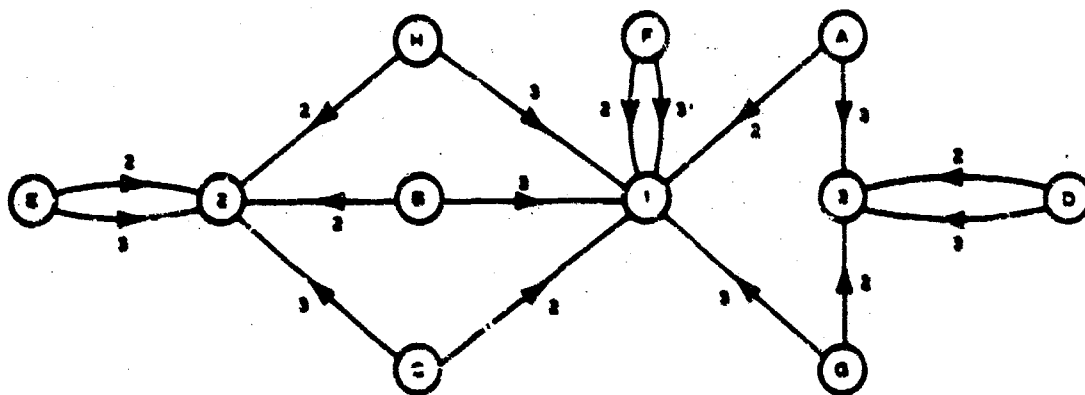


A content-address search for those items whose second and third fields are 2 and 1, respectively, is transformed to a pa    search for

Any content-address search can be similarly transformed.

## 6. STRUCTURE-ADDRESSING BY CONTENT-ADDRESSING

Given a multiple-comparand content-addressable memory, one can implement structure-addressing on it; one stores a word for each link of the graph containing the initial node label, the link weight, and the terminal node label.

As an example, the graph previously shown could be stored in a CAM as follows:

| Initial node | Link weight | Terminal node |
|:---:|:---:|:---:|
| A | 2 | 1 |
| A | 3 | 3 |
| B | 2 | 2 |
| B | 3 | 1 |
| C | 2 | 1 |
| C | 3 | 2 |
| D | 2 | 3 |
| D | 3 | 3 |
| E | 2 | 2 |
| E | 3 | 2 |
| F | 2 | 1 |
| F | 3 | 1 |
| G | 2 | 3 |
| G | 3 | 1 |
| H | 2 | 2 |
| H | 3 | 1 |

The pattern search



could be implemented as follows:

1. Find all words with 2, 2 in their second and third fields (three responses: B, 2, 2, ; E, 2, 2; H, 2, 2).

2. Form a comparand for each response whose first field is the first field of the response and whose second and third fields are 3 and 1 respectively (three comparands: B, 3, 1; E, 3, 1; H, 3, 1).

3. Find all words that agree with one of these comparands (two responses: B, 3, 1; H, 3, 1).

B and H satisfy the pattern search (the first fields of the responses to Step 3).

This example shows the use of multiple comparands. In general, they will be required in many structure searches, being formed from the responses of one step of the search for use in a later step. If the data structure is large, there may be many comparands in some step of a search; a single-comparand CAM can only treat these one at a time and may become unduly slow.

A machine organization using a single-comparand CAM for structure addressing is the Association-Storing Processor. [3] Since only one comparand is permitted at a time, the search algorithm involves a "backtrack" procedure; that is, after any step it treats one of the responses in the next step, carrying it to completion, and then treats the others in later steps. The time spent in a given search depends strongly on the complexity of the data structure being searched; some structures may generate numerous responses and hence, numerous backtrackings.

From these thoughts, the desirability of a multiple-comparand content-addressed memory can be seen.

## 7. A SORTING MEMORY AS A MULTICOMPARAND CAM

### a. General

One way to build a multicomparand CAM is to use multiple response stores, one for each comparand. The response store in a CAM is a major cost item and thus this solution is uneconomical. Another way is to use a sorting memory (Append. VI). This has the advantage that the cost increment is small as comparands are added. One limitation is that only searches on the left parts of words can be performed; proper organization of data removes the effect of this limitation.

### b. Main Section

A sorting memory used for multiple-comparand content-addressing has 11 different words in its main section. Their formats are shown in Figure XVI-1. The leftmost bits of these words are 0. The high end of the sorting memory is the readout section and contains words with leftmost bits equal to 1. The readout section is discussed in Item c below.

Empty words contain all zeros except for two bits as shown in Figure XVI-1. Their magnitude is less than that of any other word in memory and thus they collect at the low end of memory. The low end is used for input so the empty words are overwritten with new data or operations through the input lines.

Each link of the data structure is represented by two link words - a forward word and a backward word. The node labels are interchanged and the bit between the A and B fields is changed from 0 to 1 in the backward word. Because of the sorting action, the forward words of all links leaving a given node are collected and ordered by their weights. Similarly, the backward words of all links entering the node are collected in a set adjacent to the forward words of links leaving the node.
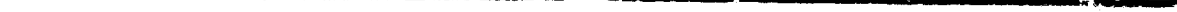
Figure XVI-1 · Word Formats in Multicomparand Content-Addressed Sorting Memory

Operations 1 and 2 (Figure XVI-1) destroy a word with a given link weight and node labels. They are characterized by a 1 between the B and C fields and a 0 in the rightmost position. The bit between the A and B fields indicates which operation is to be done, forward word destruction or backward word destruction. In either case, the sorting action sends the operation word to a location just below the word to be destroyed. Circuitry in the memory detects the existence of the operation word and changes it and its corresponding link word to empty words. If no link word corresponds, the operation word alone is destroyed.

Operations 3 and 4 read all the links on a given node. A 0 between the B and C fields and a 11 in the two leftmost C-field positions characterize these operations. The sorting action sends these operation words to a location just below the links of the given node. Circuitry in the memory detects the presence of one of these operation words and causes the following action:

1.   The C-field contents of the operation word replaces the A field of all corresponding link words.

2.   The A- and C-field contents of the operation word are interchanged.

3.   The leftmost bits of the link words and operation word are changed from 0 to 1.

This action causes the operation word and link words to travel to the readout section during the next sort cycle. The link words are sent back to the main section after readout if and only if the rightmost operation word bit is 1 (Operation 3 rather than Operation 4).

Operations 5, 6, 7, and 8 read all links with a given weight and direction incident on a given node. A 0 between the B and C fields and a 01 in the two leftmost C-field positions characterize these operation words. The sorting action sends the operation words

to a location just below the links to be read.   Circuitry
in memory detects their presence and causes exactly the same
action as that for Operations 3 and 4; only the link words with the de-
sired link weight are treated.

c.   Readout Section

As indicated in b above, Operations 3 through 8 cause the leftmost
bits of certain words to be changed from 0 to 1; during the succeeding
sort cycle, these words arrive at the high end of memory (the read-
out section).   At any time, each of the operation words has a unique
control so that no intermingling of responses between concurrent
operations can occur (the control is put in the A field as discussed
in b above).

Readout lines connected to the high end of memory read the contents
of the readout section after which circuitry in memory causes the
following action:

  1.   Any link word associated with an operation word
       whose rightmost bit is 0 is overwritten with an
       empty word (this destroys this link word).

  2.   The A-field of any link word associated with an
       operation word whose rightmost bit is 1 is re-
       placed by the C field of the operation word (this
       was its original A field) and the leftmost bit of
       this link word is changed back to 0 (this puts it
       back in its original state).

  3.   Any operation word is overwritten with an empty
       word.

In the succeeding sort cycle all undestroyed link words return to their
former positions in the main section of memory.

d.   Conflicts between Operation Words

There is a possibility that more than one operation word wants to

affect the same link words. These conflicts are detected in the main section and resolved with the following rules:

1. Operations 1 and 2 take precedence over the other operations. Any other operation can still read any link not being destroyed by an Operation 1 or 2.

2. Operations 5, 6, 7, and 8 take precedence over Operations 3 and 4. Otherwise, conflicts are resolved in favor of the operation word with highest control field. Any operation word losing out to another by this rule is "delayed" as discussed below.

An operation word losing a conflict by Rule 2 is delayed by changing the 1 in its second leftmost C-field position to a 0. The word remains in this state during the succeeding sort cycle. At the end of this cycle, the bit is changed back to a 1 but no other action occurs until the end of the following sort cycle at which time the operation is tried again (at this time any undestroyed links that were sent to the readout section by the operation winning the conflict have been returned to their original state).

Operations 3 through 8 cause link words to be absent from the main section for two sort cycles. A simple rule can prevent the possibility that an operation word arrives in the main section while the links it wants are in the readout section. The rule is to input operation words with odd A fields only during alternate cycles and operation words with even A fields only during the intervening cycles.

The restriction mentioned in the foregoing paragraph can be removed by the addition of "place marker" words to those of Figure XVI-1. Such words would remain in the main section in place of those links sent to the readout section.

## 8. A PARALLEL NONNUMERIC PROCESSOR

A parallel nonnumeric processor could be constructed using the multiple-comparand content-addressed sorting memory described in Item 7. Figure XVI-2 shows a block diagram of such a processor.

In general, the processing unit sends operation words and new link words to the memory and receives responses in return. The control fields originally entered in the operation words wind up in the responses so that no ambiguity occurs even though many different operation words may be present. The control fields are used in the processing unit to send the responses to the correct locations. Input and output channels communicate with the processing unit. These can be handled in a manner similar to that described in Appendix XV.

Further development of the processing unit is dependent on development of general-purpose structure-search algorithms. The basic form for an algorithm that treats both searches with loops and "loop-free" searches is discussed below. Arithmetic and other operations need also be included to obtain a useful machine.
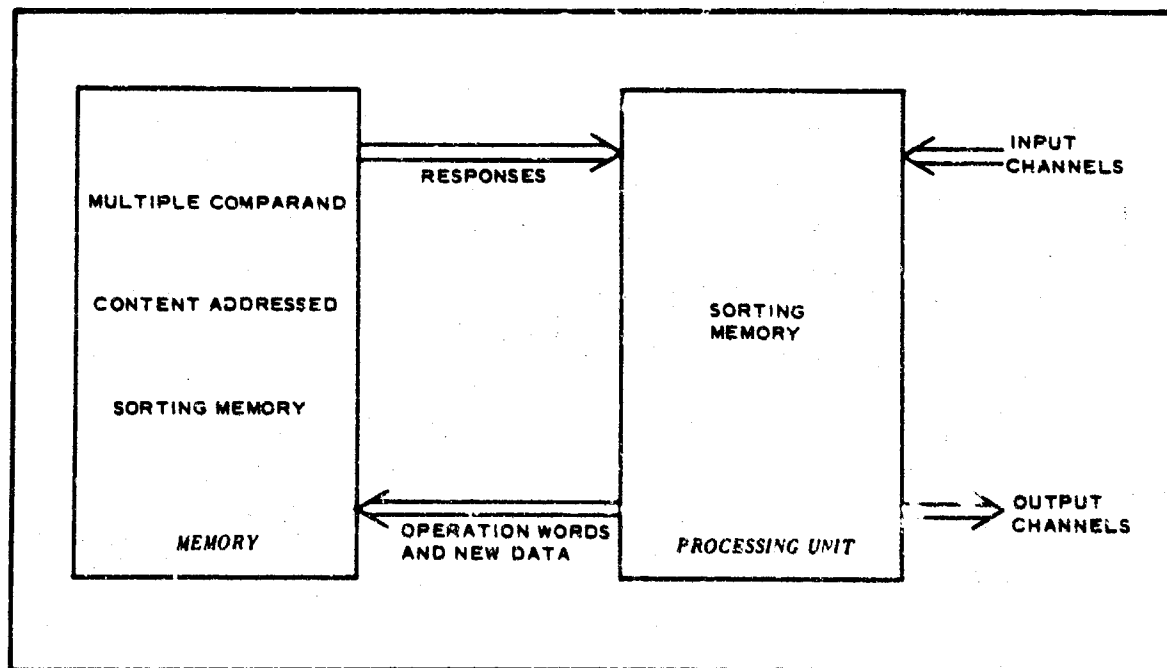


Figure XVI-2 - A Parallel Nonnumeric Processor

## 9.   ALGORITHM FOR PARALLEL-STRUCTURE SEARCHES

This algorithm will search any data structure for a subpattern that meets
the following conditions:

      1.   All link weights in the search pattern are constant
           (have known weights), and

      2.   At least one node in the search pattern is constant
           (has a known label).

The algorithm works in "parallel," treating all possible search candidates
simultaneously.   The time in most cases is proportional to the number of
links in the search pattern unless storage limits are reached.

The algorithm produces a set of n-tuples where n is the number of vari-
ables in the search pattern.   Depending on the implementation of the al-
gorithm, the n-tuples might be stored as ordered sets of words in the
processing unit or the n-tuples might be represented in memory; for ex-
ample, a new node for each n-tuple connected to a fixed node and to all of
its members by links with certain weights.

We assume the variable nodes of the search pattern are labelled by $X_1$,
$X_2, \ldots, X_n$.   The ith term (for $1 \leq i \leq n$) of each final n-tuple will
contain the node label of $X_i$ in the subpattern corresponding to the n-tuple.

With no loss of generality, it can be assumed that for any pair of variables
in the search pattern there exists at least one path between them incident
only on variable nodes.   If this condition is not met, then the search pat-
tern can be split along some of its constants into two or more disconnected
pieces; each of the pieces meets the condition and can be treated inde-
pendently of the other pieces.   Furthermore, any link between constants
is redundant and can be removed.

The only housekeeping required is a method of marking treated links in
the search pattern to distinguish them from untreated links.

The algorithm is as follows:

Step 1. Pick any link in the search pattern incident on a constant. The other node of the link is a variable, say $X_i$. Depending on link direction, send an Operation 5 or 7 word to memory (see Figure XVI-1) using the constant and link weight in fields A and B, respectively. The responses are candidates for $X_i$. Form an n-tuple for each response with the response node label as its ith member. Mark the search pattern link as being treated. Go to Step 2.

Step 2. Is there any untreated link in the search pattern between a constant and a variable incident on a treated link? If so, go to Step 3; otherwise, go to Step 4.

Step 3. Let the untreated link of Step 2 have link weight W, constant node C, and variable node $X_i$. Depending on link direction send an Operation 5 or 7 word to memory with C and W in fields A and B, respectively. Compare the responses to the ith members of all n-tuples and destroy any n-tuple whose ith member does not correspond to any response. Mark the link as being treated. Go to Step 2.

Step 4. Is there any untreated link in the search pattern between two variable nodes, each of which is incident on some treated link? If so, go to Step 5; otherwise, go to Step 6.

Step 5. Let the untreated link have weight W, initial node $X_i$, and terminal node $X_j$. For each n-tuple, send an Operation 5 word to memory with its ith member in the A field and W in its B field and discard the n-tuple if its jth member does not agree with any of the responses. Mark the link as being treated. Go to Step 4.

Step 6. Are all links in the search pattern treated? If so, the algorithm is complete; if not, pick an untreated link one of whose nodes is incident on a treated link and go to Step 7.

Step 7. Let the untreated link have weight W. Let the node incident on a treated link be $X_i$ and let the other node be $X_j$. Depending on link direction, send an Operation 5 or 7 word to memory for each n-tuple. The A field of the operation word is the ith member of the n-tuple and the B field is W. If the operation word for the n-tuple has m responses, replicate the n-tuple m times and put one response in the jth member of each copy. Mark the link as being treated. Go to Step 2.

This algorithm can be halted after Steps 1, 3, 5, or 7 if no n-tuples are present. This condition means that no subpattern of the data structure matches the search pattern.

## 10. CONCLUSIONS

This appendix has discussed nonnumeric processing in general and characteristics that are present in current machines and also those characteristics desirable in a parallel nonnumeric processor. It was shown that a multicomparand CAM exhibits these desirable characteristics. It was further shown that a sorting memory can serve as an implementation of a multicomparand CAM. A general form for a parallel nonnumeric processor was described. A basic search algorithm was presented for parallel structure searches.

Time limitations prevented development of the detailed processor design. A general search algorithm should also be developed. However, study to date indicates that a machine patterned after the organization in this report would be capable of solving large nonnumeric problems significantly faster than other existing schemes.

## 11. LIST OF REFERENCES

1. McCormick, B. H.: The Illinois Pattern-Recognition Computer (Illiac III). Urbana, Ill., University of Illinois, Digital Computer Laboratory Report No. 148, 1963.

2. Slotnick, D. L., et al.: "SOLOMON." Proceedings of The Fall Joint Computer Conference, 1962.

3. RADC-TR-65-32: Association-Storing Processor. Rome Air Development Center, Griffiss Air Force Base, N.Y.

Security Classification

## DOCUMENT CONTROL DATA - R&D

*(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)*

| 1. ORIGINATING ACTIVITY (Corporate author) | 2a. REPORT SECURITY CLASSIFICATION |
|---|---|
| Goodyear Aerospace Corp | Unclassified |
| | 2b. GROUP |

3. REPORT TITLE

Advanced Computer Organization Study
Volumes I and II

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Final Report   August 1964 - November 1965

5. AUTHOR(S) (Last name, first name, initial)

Rohrbacher, Donald L.

| 6. REPORT DATE | 7a. TOTAL NO. OF PAGES | 7b. NO. OF REFS |
|---|---|---|
| April 1966 | 558 | 4 |

| 8a. CONTRACT OR GRANT NR. | 9a. ORIGINATOR'S REPORT NUMBER(S) |
|---|---|
| AF30(602)-3550 | |
| b. PROJECT NO. | GER-12314 |
| 4594 | |
| c. Task 459406 | 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) |
| d. | RADC-TR-66-7 |

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

| 11. SUPPLEMENTARY NOTES | 12. SPONSORING MILITARY ACTIVITY |
|---|---|
| | Rome Air Development Center GAFB, N.Y. 13440. |

13. ABSTRACT Advanced general-purpose computer organizations capable of parallel data processing were studied. To achieve maximum system performance from highly parallel computer organizations, new solution models and programming techniques must be developed. Hence, the following three areas were investigated simultaneously:
1. Applications - Study of problems and their inherent degree of parallelism, and development of theoretical solution models for use on a parallel processor.
2. Programming - The programming of parallel solution models on the postulated computer organizations.
3. Machine Organization - Development of machine implementations capable of parallel data processing.
This study resulted in the design of two computer organizations (designated Machine I and Machine II) capable of parallel data processing and fast sorting and table searching in memory. These machine organizations were possible because of the development of a special memory that permits many processing and input-output units to access memory simultaneously without conflict.
The applications effort was focused on the development of solution models which exploited the maximum amount of parallelism resident within a problem. Two major problems were investigated: a dynamic programming problem, and parallel compilation.
Detailed programs were written for the dynamic programming problem on Machine I and a parallel compilation algorithm on Machine II. These same problems also were programmed on the IBM 7090 to provide a standard of comparison. In both cases, the parallel processing capability of the machines afforded significant increases in speed of program execution.

DD FORM 1 JAN 64 1473

| 14. KEY WORDS | LINK A | | LINK B | | LINK C | |
|---|---|---|---|---|---|---|
| | ROLE | WT | ROLE | WT | ROLE | WT |
| Computer<br>Programming<br>Numerical Analysis | | | | | | |

## INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization (corporate author) issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers (either by the originator or by the sponsor), also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through
_____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through
_____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through
_____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring (paying for) the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as (TS), (S), (C), or (U).

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.